SYBEX

Programmierung des 8086/8088

J. W. Coffron

Programmierung des 8086/8088

Programmierung des 8086/8088

James W. Coffron

SYBEX
DÜSSELDORF
BERKELEY · LONDON · PARIS

Anmerkungen:

8086 und 8088 sind Warenzeichen von Intel Corporation.
Alle Mnemonics sind urheberrechtlich geschützt durch Intel Corporation.
IBM PC ist ein geschütztes Warenzeichen von International Business Machines, Inc.

Originalausgabe in Englisch.
Titel der englischen Ausgabe: PROGRAMMING THE 8086/8088
Original Copyright © 1983 by SYBEX Inc., Berkeley, California, USA

Deutsche Übersetzung: Alfons Steinhoff, Dortmund

Umschlag: Jean-François Penichoux
Technische Zeichnungen: Ingrid Owen
Satz: tgr – typo-grafik-repro gmbh, Remscheid
Gesamtherstellung: Druckerei Hub. Hoch, Düsseldorf

Der Verlag hat alle Sorgfalt walten lassen, um vollständige und akkurate Informationen zu publizieren. SYBEX-Verlag GmbH, Düsseldorf, übernimmt keine Verantwortung für die Nutzung dieser Informationen, auch nicht für die Verletzung von Patent- und anderen Rechten Dritter, die daraus resultieren.

Alle deutschsprachigen Rechte vorbehalten. Kein Teil des Werkes darf in irgendeiner Form (Druck, Fotokopie, Mikrofilm oder einem anderen Verfahren) ohne schriftliche Genehmigung des Verlages reproduziert oder unter Verwendung elektronischer Systeme verarbeitet, vervielfältigt oder verbreitet werden.

ISBN 3-88745-050-7
1. Auflage 1984
2. Auflage 1985
3. Auflage 1985

Printed in Germany
Copyright © 1984 by SYBEX-Verlag GmbH, Düsseldorf

Inhaltsverzeichnis

Einführung 7

1 Grundlagen **11**
Was ist Programmierung? 11
Flußdiagramme 12
Darstellung von Informationen 14

2 Der innere Aufbau des 8086/8088 . . . . . **37**
Allgemeine Mikroprozessor-Systemarchitektur 37
Das Innere eines Mikroprozessors 40
Der allgemeine Befehlszyklus 44
Die interne Organisation des 8086/8088 47
Die allgemeinen Register des 8086/8088 49
Zusätzliche Kontrollkennzeichen 53
Zusammenfassung 54

3 Die Speicherorganisation und Adressierungsarten des 8086/8088 . . . . . **55**
Die Systemspeicherorganisation des 8086/8088 55
Ausgerichtete und nichtausgerichtete Wörter 59
Systemspeicherorganisation des 8088 60
Adreßerzeugung mit dem 8086/8088 61
8086/8088-Objektcode 66
Objektcode für Indizierung und Benutzung von Basisregistern . 70
Zusammenfassung Objektcode 70
Zusammenfassung 71

4 Der Befehlssatz: Individuelle Beschreibungen **73**

5 Grundlegende Programmierungstechniken **175**
Arithmetische Programme 175
BCD-Arithmetik 181
Unterprogramme 188
CALL-RETURN-Unstimmigkeit 192
Zusammenfassung 194

6 Unterbrechungen bei dem 8086/8088 . . . . . **195**
Was ist eine Unterbrechung 195
CPU-Tätigkeit während der NMI-Anforderung 198

Die Unterbrechungstabelle 199
INTR-Eingabe 202
Interne Unterbrechungen 204
Das Zurücksetzen (RESET) des 8086/8088 207
Zusammenfassung 208

7 Eingabe und Ausgabe für den 8086/8088 . . . . . **209**
Was sind Eingabe und Ausgabe 209
E/A-Adressierung 211
Eingabebefehl 214
Ausgabebefehl 216
8086-Eingabe- und Ausgabeanschlüsse 217
8088-Eingabe- und Ausgabeanschlüsse 217
Die Benutzung eines 8255-PIO-Bausteins mit dem 8086/8088 . . 223
Ein Beispiel für das Abtasten eines Tastenfeldes 231
Zusammenfassung 237

8 Weitere Anwendungen mit dem IBM Personal Computer **239**
BIOS (Basic I/0 System, Grund-E/A-System 239
Vorteile des BIOS 240
Der Zugriff auf Peripheriegeräte mit dem BIOS 241
Präambel für die Drucker-E/A 243
Tastatur-Präambel 252
RS-232-Präambel 253
Programmdefinition 255
Zusammenfassung 259

9 Programmentwicklung **261**
Programmierungsmöglichkeiten 261
Die Stufen der Softwareentwicklung 267
Hardwarealternativen 268
Zusammenfassung der Hardwarebetriebsmittel 269
Der Assembler 270
Die Assemblersprache 273
Direktiven zur Verwaltung des segmentierten Speichers 278
Zusammenfassung 280
Schlußbemerkung 280

Anhang A: Umwandlungstabellen Hexadezimal – Dezimal 281
Anhang B: ASCII-Tabelle 283
Anhang C: Umwandlungstabelle BCD – Dezimal 285
Anhang D: Referenzdaten zum Befehlssatz des 8086/88 287
Stichwortverzeichnis 303

Einführung

Willkommen in der erregenden Welt der 16-Bit-Mikroprozessoren. Stellen Sie sich vor, Sie selbst programmieren, steuern und benutzen einen der leistungsstärksten und vielseitigsten 16-Bit-Mikroprozessoren: den 8086. Dieser Mikroprozessor hat Zugriff auf über EINE MILLION Speicherzellen – weit entfernt von den verfügbaren ca. 64 000 der Mehrzahl der 8-Bit-Mikroprozessoren.

Der 8086 ist einer der in der Industrie heute am weitesten verbreiteten 16-Bit-Mikroprozessoren. Er hat einen extrem leistungsfähigen Befehlssatz, der es Ihnen gestattet, Ihre eigene Software-Anwendung zu realisieren. Beispielsweise hat der 8086 zwei Instruktionen, die es Ihnen erlauben, Zahlen direkt zu multiplizieren und zu dividieren; so werden keine aufwendigen Algorithmen mehr benötigt, um diese Aufgaben durchzuführen.

Während Sie lernen, den 8086 zu programmieren, lernen Sie gleichzeitig, den 8088-Mikroprozessor zu programmieren. Diese beiden Prozessoren haben einen identischen Befehlssatz, und Software, die für einen der beiden geschrieben wurde, läuft auf dem anderen ohne Änderungen.

Der 8086 und der 8088 sind beide gut geeignet für die Verbindung mit E/A-Geräten und E/A-Anschlüssen. So kann ihre Architektur mehr als 64 000 einzelne Eingabe- und Ausgabeanschlüsse unterstützen. Diese Anschlüsse können irgendeine Kombination von 8 oder 16 Bits sein!

Zusätzlich bieten der 8086 und der 8088 ein leistungsstarkes Konzept zur Unterbrechungsbehandlung, das Ihnen erlaubt, jede Systemadresse als Unterbrechungsvektor für Dienstroutinen zu benutzen. Es gibt ferner spezielle Unterbrechungsvektoren für Einzelschrittverarbeitung und für interne Fehler wie z. B. Division durch Null. Überdies kann jede Unterbrechungsbehandlungsroutine durch Hardware oder Software ausgelöst werden.

Der 8086 und der 8088 wurden für den problemlosen Einsatz in Multiprozessorsystemen entworfen. In diesem Sinne hat die Firma INTEL verschiedene Koprozessoren für den Gebrauch mit dem 8086 und 8088 entworfen.

Die vorangehenden Punkte und noch viele andere Dinge mehr werden in diesem Buch umfassend behandelt. Sie werden die Programmierung des 8086/8088 gründlich lernen. Vom inneren Aufbau des Mikroprozessors

bis hin zu den hochentwickelten Adressierungsarten wird Ihnen gezeigt, wie alles funktioniert. Jeder Befehl wird beschrieben, und mehrere Beispiele für tatsächliche Programme werden angegeben. Um den Erklärungen mehr Realitätsnähe zu geben, stellen wir Programmbeispiele vor, die für den IBM PC geschrieben sind, der den 8088-Mikroprozessor verwendet.

Deshalb wird Ihnen dieses Buch von Nutzen sein, ob Sie ein Anfänger im Programmieren von Mikroprozessor-Assemblersprache oder ein fortgeschrittener Programmierer sind. Nach der Lektüre dieses Buches sollen Sie in der Lage sein, den 8086/8088 für jede beliebige Anwendung zu programmieren. Außerdem soll dieses Buch später als wertvolles Nachschlagewerk dienen.

Auf geht's! Wir wollen gemeinsam den Weg beschreiten, der Sie zum Verständnis der Leistungsmöglichkeiten führt, die in einem der höchstentwickelten 16-Bit-Mikroprozessoren unserer Zeit stecken.

Kapitelabriß

Kapitel 1 beginnt mit einer Übersicht über die grundlegenden Konzepte der Mikroprozessorprogrammierung. Es beinhaltet die Zahlendarstellung, die Addition und Subtraktion von binären und BCD-codierten Zahlen und den Gebrauch von Kennzeichenbits.

Kapitel 2 beschreibt allgemein den inneren Aufbau eines Mikroprozessors und erörtert insbesondere die internen Register des 8086/8088. Eine in die Tiefe gehende Betrachtung konzentriert sich auf die Unterschiede und Ähnlichkeiten zwischen dem 8086 und dem 8088 und Mikroprozessoren im allgemeinen.

Kapitel 3 beschreibt detailliert die physikalische Speicherorganisation des 8086 und des 8088 und zeigt, daß sie völlig verschieden sind. Zusätzlich untersucht dieses Kapitel unterschiedliche Adressierungsarten für die verschiedenen Befehle.

Kapitel 4 ist ziemlich umfangreich. Es gibt eine vollständige Aufzählung sämtlicher Befehle des 8086/8088 und liefert für jeden Befehl wichtige Informationen.

Kapitel 5 erörtert die grundlegenden Programmiertechniken für Mikroprozessoren – Addition, Subtraktion, Multiplikation und Division – und zeigt, wie diese Techniken mit den Befehlen des 8086/8088 verwirklicht werden.

Kapitel 6 behandelt das Thema von Unterbrechungen für ein Mikroprozessorsystem im allgemeinen und beschreibt die vollständige Unterbre-

chungsstruktur für den 8086/8088. Am Ende dieses Kapitels werden Sie verstanden haben, wie Unterbrechungen mit diesen CPUs benutzt werden.

Kapitel 7 beinhaltet die Ein-/Ausgabetechniken für den 8086/8088. Verschiedene Eingabe und Ausgabebefehle werden diskutiert, und es wird untersucht, wie sie einzusetzen sind. Eingeschlossen sind Programme zur Steuerung einer 8255-PIO und eines 8253-Zähler/Zeitgeber-LSI-Bausteins.

Kapitel 8 behandelt die Anwendungen in der „realen Welt". Es zeigt Ihnen, wie Sie Ihr Wissen über den 8088 zur Steuerung des IBM PC benutzen können. Es werden zwei verschiedene Anwendungsprogramme angegeben. Jedes benutzt das IBM BIOS zur Steuerung des Druckers, der Tastatur und des RS-232-Anschlusses.

Kapitel 9 untersucht verschiedene Techniken zur Programmentwicklung. Entwicklungssysteme werden bewertet und Entscheidungshilfen werden gegeben. Das Kapitel schließt mit einer Diskussion von Assemblersprachendetails ab, die Sie brauchen, um Programme für die 8086/8088-CPU zu schreiben.

Weiterhin sind noch vier nützliche Anhänge vorhanden. *Anhang A* ist eine dezimal-hexadezimale Umwandlungstabelle. *Anhang B* ist eine ASCII-dezimal-Tabelle. *Anhang C* ist eine dezimal-zu-BCD-Umwandlungstabelle. *Anhang D* ist eine Zusammenfassung des 8086/8088-Befehlssatzes.

Kapitel 1
Grundlagen

Wir beginnen mit der Behandlung der grundlegenden Konzepte und Definitionen, die in der Computerprogrammierung benutzt werden. Wenn Sie schon mit diesen Konzepten vertraut sind, möchten Sie möglicherweise nur einen flüchtigen Blick auf den Inhalt dieses Kapitels werfen, um dann mit Kapitel 2 fortzufahren. Wir schlagen jedoch vor, dieses Kapitel durchzulesen, selbst wenn Sie ein erfahrener Programmierer sind, damit Sie sich mit dem in diesem Buch gewählten Ansatz vertraut machen.

WAS IST PROGRAMMIERUNG?

Hat man ein Problem, so versucht man normalerweise, sich eine Lösung auszudenken. Eine solche Lösung bezeichnet man als Algorithmus, wenn sie in Form einer Schritt-für-Schritt-Vorschrift angegeben ist. Ein Algorithmus kann in einer beliebigen Sprache bzw. Symbolsprache formuliert werden. Er muß nach einer endlichen Anzahl von Schritten abbrechen. Hier ein einfaches Beispiel für einen Algorithmus zum Aufschließen und Öffnen einer Tür:

1. Stecke den Schlüssel in das Schlüsselloch
2. Drehe den Schlüssel eine volle Umdrehung nach links
3. Umfasse die Türklinke
4. Drücke die Türklinke herunter und stoße die Tür auf

Nun wird sich die Tür öffnen, wenn der Algorithmus für den Typ des vorliegenden Schlosses geeignet ist.

Liegt eine Lösung eines Problems in Form eines Algorithmus vor, so besteht der nächste Schritt darin, den Algorithmus in eine Sprache zu übersetzen, die vom Computer verstanden wird.

Bis heute wird nur eine genau definierte Teilmenge einer natürlichen Sprache – *Programmiersprache* genannt – von Computern „verstanden". Die Umsetzung eines Algorithmus in eine Folge von Befehlen einer Programmiersprache heißt *Programmierung*. Die eigentliche Umset-

zungsphase eines Algorithmus in die Programmiersprache heißt *Codierung*. Programmierung bezieht sich nicht allein auf die Kodierung, sondern auch auf den umfassenden Entwurf der Programme und „Datenstrukturen", die den Algorithmus ausmachen.

Effiziente Programmierung erfordert nicht nur das Verständnis möglicher Implementierungstechniken für Standardalgorithmen, sondern auch den geschulten Umgang mit Computer-Hardware-Komponenten (wie internen Registern, Speichern und Peripheriegeräten) sowie den geschickten Gebrauch passender Datenstrukturen. (Diese Techniken werden wir in den folgenden Kapiteln untersuchen.)

Programmierung erfordert ebenso eine strenge Disziplin bei der Dokumentation. Gut dokumentierte Programme sind später nicht nur dem Autor verständlich, sondern auch anderen Benutzern. Die Dokumentation muß sowohl innerhalb als auch außerhalb eines Programms erfolgen. Unter *interner Programmdokumentation* versteht man die Kommentare innerhalb des Programmtextes, die die Arbeitsweise des Programms erläutern. Die *externe Dokumentation* besteht aus den Entwurfsdokumenten, die neben dem Programm existieren, wie schriftlichen Erläuterungen, Handbüchern und Flußdiagrammen.

Ein Zwischenschritt wird fast immer zwischen den Entwurf des *Algorithmus* und die Erstellung des *Programms* geschoben. Dieser Schritt ist die *Anfertigung von Flußdiagrammen*.

FLUSSDIAGRAMME

Ein Flußdiagramm ist einfach eine symbolische Darstellung eines Algorithmus, die üblicherweise aus einer Folge von Rechtecken und Rauten besteht. In einem Flußdiagramm werden Rechtecke benutzt, um *Kommandos* (oder ausführbare Anweisungen) darzustellen. Rauten werden für Abfragen benutzt, wie z. B.: Wenn die Information X wahr ist, dann führe die Handlung A aus; wenn nicht, führe die Handlung B aus. Abbildung 1.1 ist ein Beispiel für ein Flußdiagramm. An dieser Stelle wollen wir keine formale Definition von Flußdiagrammen angeben. Wir werden die Anfertigung von Flußdiagrammen ausführlicher in Kapitel 3 besprechen.

Die Anfertigung eines Flußdiagramms ist ein sehr empfehlenswerter Zwischenschritt zwischen der Erstellung eines Algorithmus und der eigentlichen Codierung einer Lösung. Bemerkenswerterweise zeigen Beobachtungen, daß nur etwa 10% der Programmierer in der Lage sind, erfolgreich ein Programm zu schreiben, ohne ein Flußdiagramm anfertigen zu müssen. Leider weiß man auch auch, daß 90% dieser Leute glauben, zu eben diesen 10% zu gehören! Das Ergebnis ist dann, daß im Durchschnitt 80% der Programme beim ersten Lauf auf einem Computer versagen. (Diese Prozentangaben erheben natürlich keinen Anspruch darauf, exakt

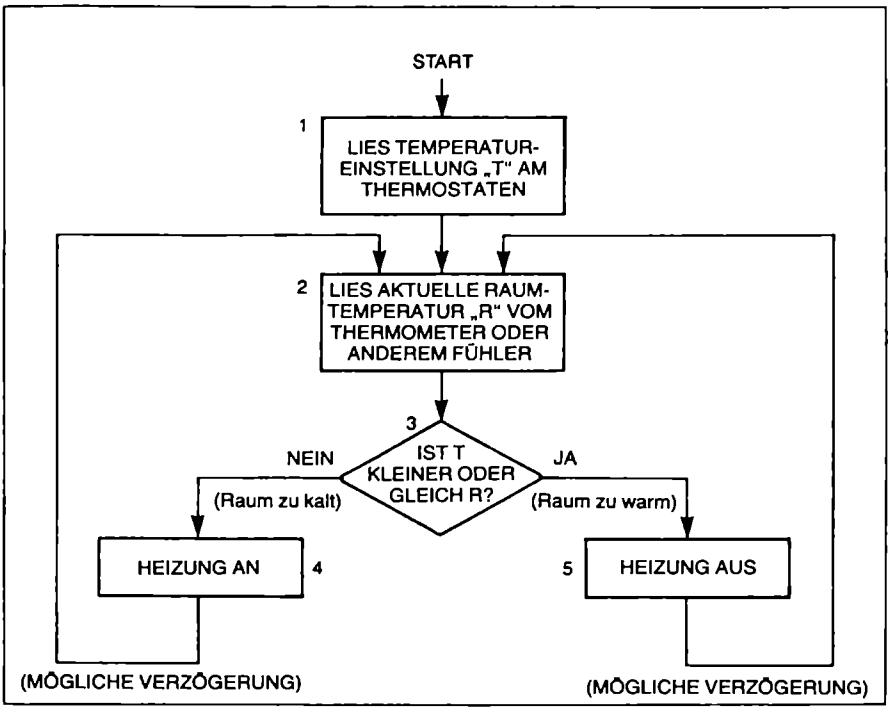


Abb. 1.1: Ein Flußdiagramm zur Regelung der Raumtemperatur

zu sein.) Kurz gesagt erkennen gerade die absoluten Anfänger im Programmieren selten die Notwendigkeit, ein Flußdiagramm zu zeichnen. Das Fortlassen des Flußdiagramms hat häufig ein fehlerhaftes Programm zur Folge, und der Programmierer (oder die Programmiererin) muß dann viel Zeit zum Testen und Korrigieren seines (oder ihres) Programms verwenden. (Dies nennt man die Testphase, engl. *debugging*.) Die Methode der Flußdiagramme wird deshalb für alle Fälle sehr empfohlen. Die Anfertigung eines Flußdiagramms erfordert im allgemeinen einen geringen zusätzlichen Zeitaufwand vor der Codierung und führt üblicherweise zu einem übersichtlichen Programm, das innerhalb kurzer Zeit korrekt läuft. Wenn diese Technik einmal richtig verstanden ist, kann ein geringer Prozentsatz von Programmierern diesen Schritt in Gedanken, ohne Papier zu benutzen, vollziehen. Leider sind in solchen Fällen die Programme für andere gewöhnlich schwer zu verstehen, da der Teil der Dokumentation, den das Flußdiagramm darstellt, nicht zur Verfügung steht. Es wird daher generell empfohlen, die Anfertigung von Flußdiagrammen als strenge Vorschrift für alle Programme zu beachten, die aus mehr als zehn bis fünfzehn Befehlen bestehen. In diesem Buch werden viele Beispiele für Flußdiagramme angegeben.

DARSTELLUNG VON INFORMATION

Alle Computer verarbeiten Informationen in Form von Zahlen oder Buchstaben. Wir werden nun die externe und die interne Darstellung von Information auf einem Computer untersuchen.

Interne Darstellung von Information

Information wird in einem Computer in Gruppen von Bits gespeichert. Das Wort *Bit* steht für engl. *binary digit*, was soviel wie zweiwertige Ziffer bedeutet. Aufgrund der Beschränkungen konventioneller Elektronik benutzt die praktikabelste Darstellung von Information eine Logik mit zwei Zuständen. Die beiden Zustände der Schaltkreise, die in der digitalen Elektronik benutzt werden, sind im allgemeinen „Ein" und „Aus". Diese Zustände werden logisch durch die Symbole „0" und „1" dargestellt. Da diese Schaltkreise dazu benutzt werden, logische Funktionen auszuführen, werden sie *binäre logische Schaltkreise* genannt. Das führt dazu, daß heute praktisch die gesamte Informationsverarbeitung im binären Format durchgeführt wird. Eine Gruppe von acht Bits wird *Byte* genannt. Eine Gruppe von vier Bits wird *Nibble* genannt.

Nun wollen wir sehen, wie Informationen in diesem binären Format intern dargestellt werden. Zwei Dinge müssen innerhalb des Computers dargestellt werden: das Programm (das eine Folge von Befehlen ist) und die Daten, mit denen das Programm arbeitet. Diese Daten können Zahlen oder alphanumerischer Text sein. Wir wollen nun die Darstellung von Befehlen, Zahlen und alphanumerischen Zeichen besprechen.

Viele der folgenden Beispiele benutzen acht Bits. Dies erleichtert es, das betrachtete Konzept zu erläutern. Die Beispiele können leicht auf mehr als acht Bits erweitert werden.

Darstellung numerischer Daten

Die Darstellung von Zahlen im binären Format ist keine einfache Aufgabe: verschiedene Fälle sind zu unterscheiden. Wir müssen in der Lage sein, ganze Zahlen darzustellen, Zahlen mit Vorzeichen (das sind positive oder negative Zahlen, auch als Integerzahlen bezeichnet) und Zahlen mit Dezimalpunkt. Wir wollen nun diese Anforderungen und mögliche Lösungen angehen.

Ganze Zahlen können *direkt binär* dargestellt werden. Diese direkte binäre Codierung ist einfach die Darstellung des dezimalen Wertes einer Zahl im binären System. Im binären System repräsentiert das am weitesten rechts stehende Bit 2 hoch 0. Das nächste Bit links davon repräsentiert 2 hoch 1; das folgende 2 hoch 2; und das am weitesten links stehende Bit 2 hoch 7, was gleich 128 ist. Zum Beispiel stellt

$$b_7b_6b_5b_4b_3b_2b_1b_0$$

den Wert

$$b_7 2^7 + b_6 2^6 + b_5 2^5 + b_4 2^4 + b_3 2^3 + b_2 2^2 + b_1 2^1 + b_0 2^0$$

dar. Die Potenzen von 2 sind:

$$2^7 = 128,\ 2^6 = 64,\ 2^5 = 32,\ 2^4 = 16,\ 2^3 = 8,\ 2^2 = 4,\ 2^1 = 2,\ 2^0 = 1$$

Die binäre Darstellung ist analog zur dezimalen Darstellung von Zahlen. Zum Beispiel setzt sich 123 so zusammen:

```
  1 x 100 = 100
+2 x  10 =  20
+3 x   1 =   3
-------------
          123
```

Beachten Sie, daß $100 = 10^2$, $10 = 10^1$, $1 = 10^0$ ist. In dieser stellenabhängigen Notation wird jeder Ziffer eine Potenz von 10 zugeordnet. Im binären System wird jeder binären Ziffer oder jedem Bit eine Potenz von 2, anstelle von 10 im dezimalen System, zugeordnet. Nun wollen wir ein Beispiel einer binären Zahl betrachten. 00001001 im binären System stellt dar:

```
1 x   1 = 1    (2^0)
0 x   2 = 0    (2^1)
0 x   4 = 0    (2^2)
1 x   8 = 8    (2^3)
0 x  16 = 0    (2^4)
0 x  32 = 0    (2^5)
0 x  64 = 0    (2^6)
0 x 128 = 0    (2^7)
-------------
    oder 9 dezimal
```

Lassen Sie uns einige andere Beispiele ansehen. 10000001 ist

```
1 x   1 =   1
0 x   2 =   0
0 x   4 =   0
0 x   8 =   0
0 x  16 =   0
0 x  32 =   0
0 x  64 =   0
1 x 128 = 128
-------------
    oder 129 dezimal
```

Demnach stellt 10000001 die Dezimalzahl 129 dar. Nach der Untersuchung der binären Zahlendarstellung ist es leicht zu verstehen, warum die Bits von rechts nach links mit 0 bis 7 durchnumeriert werden. Bit 0 ist b_0 und entspricht 2^0. Bit 1 ist b_1 und entspricht 2^1, usw. Abbildung 1.2 zeigt die binäre Darstellung der Zahlen 0 bis 255.

Dezimal-Binär-Umwandlung

Umgekehrt können wir das binäre Äquivalent der Dezimalzahl 11 berechnen:

11 : 2 = 5 Rest 1 > 1 *(niedrigstes Bit)*
5 : 2 = 2 Rest 1 > 1
2 : 2 = 1 Rest 0 > 0
1 : 2 = 0 Rest 1 > 1 *(höchstes Bit)*

Das binäre Äquivalent ist 1011 (Lesen Sie die äußerste rechte Spalte von unten nach oben). Das binäre Äquivalent einer Dezimalzahl erhält man, indem man solange durch 2 dividiert, bis der Quotient 0 ist.

Rechnen mit binären Zahlen

Die Rechenregeln für binäre Zahlen sind einfach. Die Regeln für die Addition sind:

0 + 0 = 0
0 + 1 = 1
1 + 0 = 1
1 + 1 = (1) 0

wobei (1) ein Übertrag von 1 bedeutet. (Beachten Sie, daß 10 das binäre Äquivalent der Dezimalzahl 2 ist.) Binäre Subtraktion kann durch „Addition des Komplements" durchgeführt werden. Wir werden die binäre Subtraktion besprechen, wenn wir lernen, wie negative Zahlen darzustellen sind. Lassen Sie uns folgendes Beispiel zur Addition betrachten:

dezimal	binär
2	10
+1	+01
3	11

Die Addition wird ebenso wie im Dezimalsystem durch Addition der Spalten von rechts nach links durchgeführt. Zuerst addieren Sie die äußerste rechte Spalte:

```
  0
+ 1
-----
  1 ( 1+0=1 Kein Übertrag)
```

Dann die nächste Spalte:

```
  1
+ 0
-----
  1 ( 0+1=1 Kein Übertrag)
```

DEZIMAL	BINÄR	DEZIMAL	BINÄR
0	00000000	32	00100000
1	00000001	33	00100001
2	00000010	•	
3	00000011	•	
4	00000100	•	
5	00000101	63	0011111
6	00000110	64	0100000
7	00000111	65	0100001
8	00001000	•	
9	00001001	•	
10	00001010	•	
11	00001011	127	01111111
12	00001100	128	10000000
13	00001101	129	10000001
14	00001110		
15	00001111	•	
16	00010000	•	
17	00010001	•	
•			
•			
•		254	11111110
31	00011111	255	11111111

Abb. 1.2: Dezimal-Binär-Tabelle

Weitere Beispiele zur binären Addition:

binär	dezimal	binär	dezimal
0010	2	0011	3
+ 0001	+ 1	+ 0001	+ 1
0011	3	0100	4

Das letzte Beispiel verdeutlicht die Rolle des Übertrags. Ein Blick auf die äußerst rechten Bits: 1+1=(1)0. Es entsteht ein Übertrag von 1, der zu den nächsten Bits addiert werden muß:

001 *(Spalte 0 ist gerade addiert worden)*
+ 000
+ 001 *(Übertrag)*
(1)0 *(wobei (1) ein neuer Übertrag nach Spalte 2 ist)*

Das Endergebnis ist 0100.

Lassen Sie uns ein anderes Beispiel betrachten:

binär	dezimal
0111	7
+ 0011	+ 3
1010	10

In diesem Beispiel entsteht ein weiterer Übertrag bis hin in die äußerste linke Spalte.

Mit acht Bits ist es deshalb möglich, direkt die Zahlen 00000000 bis 11111111 (0 bis 255) darzustellen. Zwei Beschränkungen sollten jedoch sofort erkennbar werden. Erstens stellen wir positive Zahlen dar. Zweitens ist die Größe dieser Zahlen auf maximal 255 begrenzt, wenn wir nur acht Bits verwenden. Lassen Sie uns nun diese Beschränkungen direkt angehen.

Binärzahlen mit Vorzeichen

Bei der binären Darstellung mit Vorzeichen gibt das Bit ganz links das Vorzeichen der Zahl an. Üblicherweise wird 0 benutzt, um eine *positive* Zahl zu kennzeichnen und 1 wird benutzt, um eine *negative* Zahl zu kennzeichnen. Zum Beispiel stellt 11111111 den Wert −127 dar, während 01111111 den Wert +127 darstellt. Wir können nun positive und negative Zahlen darstellen, aber wir haben die maximale Größe dieser Zahlen auf 127 verringert. Als ein weiteres Beispiel entspricht 00000001 dem Wert 1 (die führende 0 ist „+"), gefolgt von 0000001=1 und 10000001 ist −1 (die führende 1 ist „−").

Nun wollen wir das Problem der *Größe* angehen. Um größere Zahlen darzustellen, ist es notwendig, eine größere Anzahl von Bits zu benutzen. Wenn wir zum Beispiel sechzehn Bits (zwei Bytes) benutzen, um Zahlen darzustellen, können wir Zahlen von −32 K bis +32 K in binärer Form mit Vorzeichen darstellen. (1 K entspricht im Computerjargon 1024.) Bit 15 wird für das Vorzeichen benutzt, und die verbleibenden 15 Bits (Bit 14 bis Bit 0) werden für die Größe benutzt: 2^{15}=32 K.

Wenn wir große ganze Zahlen darstellen wollen, ist es nötig, intern eine größere Anzahl von Bytes zu verwenden. Deshalb sehen die einfachsten Formen von BASIC und andere Sprachen nur eine sehr begrenzte Genauigkeit für ganze Zahlen vor. Auf diese Weise können sie ein kürzeres internes Format für die Zahlen, die sie verarbeiten, verwenden. Bessere Versionen von BASIC und einige andere Sprachen sehen eine größere Anzahl signifikanter Dezimalstellen vor, auf Kosten einer großen Anzahl von Bytes für jede Zahl.

Lassen Sie uns nun ein anderes Problem lösen: Die Leistungsfähigkeit in bezug auf die Rechengeschwindigkeit. Wir wollen eine Addition in binärer Darstellung mit Vorzeichen durchführen, wie wir sie gerade eingeführt haben. Wir möchten +7 und −5 addieren.

	+7 ist 00000111
	−5 ist 10000101
Die binäre Summe ist:	10001100 oder −12

Das ist nicht das richtige Ergebnis, denn das lautet +2. Wir haben die Tatsache vernachlässigt, daß besondere Operationen durchgeführt werden müssen, die vom Vorzeichen abhängen, wenn wir diese Darstellung benutzen. Dies führt zu erhöhter Komplexität und verminderter Leistungsfähigkeit. Mit anderen Worten: Die binäre Addition von Zahlen mit Vorzeichen „funktioniert nicht korrekt". Das ist ärgerlich. Natürlich muß der Computer nicht nur Information darstellen, sondern er muß auch Rechenoperationen korrekt ausführen.

Die Lösung dieses Problems heißt *ZweierKomplement*-Darstellung. Wir werden die Zweier-Komplement-Darstellung anstelle der *binären Darstellung mit Vorzeichen* benutzen. Bevor wir das Zweier- Komplement einführen, schieben wir zuerst einen Zwischenschritt ein: das *Einer-Komplement*.

Das Einer-Komplement

In der Einer-Komplement-Darstellung werden alle positiven ganzen Zahlen in ihrem korrekten binären Format dargestellt. Zum Beispiel wird +3 wie üblich durch 00000011 dargestellt. Jedoch ihr Komplement −3 erhält man, indem man jedes Bit der ursprünglichen Darstellung komplemen-

tiert. Jede 0 wird in eine 1 umgewandelt und jede 1 in eine 0. In unserem Beispiel ist 11111100 die Einer-Komplement-Darstellung von −3.

Lassen Sie uns ein anderes Beispiel betrachten:

+2 ist 00000010
−2 ist 11111101

Beachten Sie, daß positive Zahlen in dieser Darstellung links mit einer 0 beginnen und negative Zahlen mit einer 1. Als einen Test wollen wir −4 und +6 addieren:

−4 ist 11111011
+6 ist 00000110
Die Summe ist: (1)00000001

wobei (1) einen Übertrag angibt. Das richtige Ergebnis sollte +2 oder 00000010 sein.

Versuchen wir es noch einmal:

−3 ist 11111100
−2 ist 11111101
Die Summe ist: (1)11111001

oder −6 plus einem Übertrag. Das richtige Ergebnis ist −5. Die Darstellung von −5 ist 11111010. Es hat nicht funktioniert.

Diese Darstellung ermöglicht es, positive und negative Zahlen auszudrücken; das Ergebnis einer einfachen Addition ist jedoch nicht immer korrekt. Wir werden nun eine andere Art der Darstellung benutzen. Sie ist aus dem Einer-Komplement entwickelt worden und heißt Zweier-Komplement-Darstellung.

Zweier-Komplement-Darstellung

In der Zweier-Komplement-Darstellung werden positive Zahlen wie üblich binär mit Vorzeichen dargestellt, ebenso wie beim Einer- Komplement. Der Unterschied liegt in der Darstellung *negativer Zahlen*. Die Darstellung einer negativen Zahl im Zweier-Komplement erhält man, indem man zunächst das Einer-Komplement berechnet und dann *eins addiert*. Lassen Sie uns ein Beispiel untersuchen:

Beispiel +3 wird in binärer Darstellung mit Vorzeichen durch 00000011 dargestellt. Das Einer-Komplement davon ist 11111100. Das Zweier-Komplement erhält man durch Addition einer Eins. Es ist 11111101.

Wir wollen eine Addition probieren:

binär	dezimal
00000011	3
+ 00000101	+ 5
00001000	8

Das Ergebnis ist richtig.

Lassen Sie uns eine Subtraktion ausprobieren:

binär	dezimal
00000011	3
+11111011	− 5
11111110	− 2

Nun wollen wir das Ergebnis identifizieren, indem wir das Zweier-Komplement berechnen:

(Das Einer-Komplement von 11111110 ist)	00000001
(Addition von 1)	+ 1
(Deshalb ist das Zweier-Komplement)	00000010 *(oder* +2)

Das Ergebnis 11111110 entspricht also −2. Das ist richtig.

Wir haben nun Addition und Subtraktion ausprobiert, und die Ergebnisse waren richtig (ohne Berücksichtigung des Übertrags). Es scheint, daß das Zweier-Komplement funktioniert!

Wir werden nun +4 und −3 addieren (die Subtraktion wird durch Addition des Zweier-Komplements ausgeführt):

+4	ist	00000100
−3	ist	11111101
Das Ergebnis ist:		(1)00000001

Wenn wir das Übertragsbit nicht beachten, ist das Ergebnis 00000001 (d.h. 1 dezimal). Das ist richtig. Wir wollen nur feststellen, ohne einen vollständigen mathematischen Beweis zu liefern, daß diese Darstellung funktioniert. Im Zweier-Komplement ist es möglich, vorzeichenbehaftete Zahlen ohne Rücksicht auf das Vorzeichen zu addieren oder zu subtrahieren. Benutzt man die üblichen Regeln der binären Addition, so erhält man das richtige Ergebnis einschließlich des Vorzeichens. Der Übertrag wird nicht berücksichtigt. Das ist ein sehr bedeutender Vorteil. Wäre es nicht der Fall, müßten wir das Ergebnis bzgl. des Vorzeichens

+	ZWEIERKOMPLEMENT-DARSTELLUNG	−	ZWEIERKOMPLEMENT-DARSTELLUNG
+ 127	01111111	− 128	10000000
+ 126	01111110	− 127	10000001
+ 125	01111101	− 126	10000010
...		− 125	10000011
		...	
+ 65	01000001	− 65	10111111
+ 64	01000000	− 64	11000000
+ 63	00111111	− 63	11000001
...		...	
+ 33	00100001	− 33	11011111
+ 32	00100000	− 32	11100000
+ 31	00011111	− 31	11100001
...		...	
+ 17	00010001	− 17	11101111
+ 16	00010000	− 16	11110000
+ 15	00001111	− 15	11110001
+ 14	00001110	− 14	11110010
+ 13	00001101	− 13	11110011
+ 12	00001100	− 12	11110100
+ 11	00001011	− 11	11110101
+ 10	00001010	− 10	11110110
+ 9	00001001	− 9	11110111
+ 8	00001000	− 8	11111000
+ 7	00000111	− 7	11111001
+ 6	00000110	− 6	11111010
+ 5	00000101	− 5	11111011
+ 4	00000100	− 4	11111100
+ 3	00000011	− 3	11111101
+ 2	00000010	− 2	11111110
+ 1	00000001	− 1	11111111
+ 0	00000000		

Abb. 1.3: Eine Zweier-Komplement-Tabelle

jedesmal korrigieren, was eine wesentlich langsamere Additions- oder Subtraktionsgeschwindigkeit zur Folge hätte.

Der Vollständigkeit halber lassen Sie uns erwähnen, daß das Zweier-Komplement nur die üblichste Darstellung für einfachere Prozessoren, wie etwa Mikroprozessoren, ist. In komplizierteren Prozessoren werden möglicherweise andere Darstellungen benutzt, zum Beispiel das Einer-Komplement. In diesem Fall ist aber eine spezielle Schaltung erforderlich, um das Ergebnis zu korrigieren.

Von nun an werden alle Zahlen mit Vorzeichen implizit intern in Zweier-Komplement-Notation dargestellt. Eine Tafel der Zweier-Komplement-Zahlen finden Sie in Abb. 1.3.

Wir werden nun Beispiele angeben, die die Regeln für das Zweier-Komplement erklären. Insbesondere bedeutet C eine mögliche Übertrags- (oder Leih-) Bedingung. Das ist Bit 8 des Ergebnisses. V bedeutet einen Zweier-Komplement-Überlauf; d.h. es gibt an, ob sich das Vorzeichen des Ergebnisses „versehentlich" geändert hat, weil die Zahlen zu groß sind.
Das ist ein im wesentlichen interner Übertrag von Bit 6 nach Bit 7 (dem Vorzeichenbit). Das wird unten noch klar werden.

Lassen Sie uns nun die Rolle des Übertrags C und des Überlaufs V erklären.

Das Übertragsbit C

Hier ein Beispiel für einen Übertrag:

binär	dezimal
10000000	128
+ 10000001	+ 129
(1)00000001,	257

wobei (1) einen Übertrag bedeutet. Das Ergebnis erfordert ein neuntes Bit (Bit 8, da das äußerste rechte Bit als Bit 0 bezeichnet wird). Das ist das Übertragsbit.

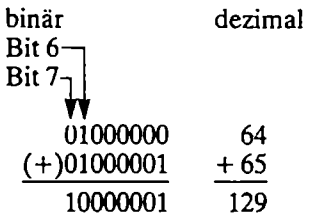
Wenn wir annehmen, daß das Übertragsbit das neunte Bit des Ergebnisses ist, erkennen wir das Ergebnis als Binärzahl 100000001 = 257. Der Übertrag muß jedoch erkannt und mit Sorgfalt behandelt werden. Im Mikroprozessor sind die Register, die zum Speichern von Informationen benutzt werden, im allgemeinen nur 8 bit lang. Wenn das Ergebnis gespeichert wird, bleiben nur die Bits 0 bis 7 erhalten.

Ein Übertrag erfordert deshalb immer besondere Handlungen. Er muß durch besondere Befehle festgestellt und dann verarbeitet werden. Die

Verarbeitung des Übertrags bedeutet, ihn entweder irgendwo zu speichern (mit einem besonderen Befehl), ihn nicht zu berücksichtigen, oder zu beschließen, daß ein Fehler vorliegt (wenn das größte zulässige Ergebnis 11111111 ist).

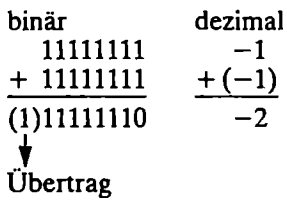
Überlauf V

Hier ein Beispiel eines Überlaufs:



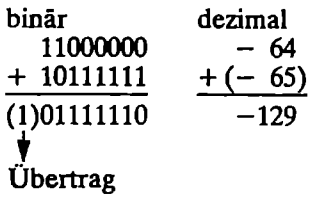
Das binäre Ergebnis 10000001 entspricht jedoch der Dezimalzahl − 127. Ein interner Übertrag ist von Bit 6 nach Bit 7 erzeugt worden. Dies nennt man einen *Überlauf*. Das Ergebnis ist nun durch ein „Versehen" negativ. Diese Situation muß erkannt werden, so daß sie korrigiert werden kann.

Lassen Sie uns eine andere Situation untersuchen:



In diesem Fall ist ein interner Übertrag von Bit 6 nach Bit 7 und auch von Bit 7 nach C erzeugt worden. Die Rechenregeln für das Zweier-Komplement besagen, daß dieser Übertrag nicht beachtet werden sollte. Das Ergebnis ist dann richtig, und zwar, weil der Übertrag von Bit 6 nach 7 das Vorzeichenbit nicht verändert hat.

Der Übertrag von Bit 6 nach 7 ist keine *Überlauf*bedingung. Wenn man mit negativen Zahlen rechnet, ist der Überlauf nicht einfach ein Übertrag von Bit 6 nach Bit 7. Lassen Sie uns ein weiteres Beispiel untersuchen:



Das binäre Ergebnis 01111111 entspricht jedoch der Dezimalzahl +127.

Dieses Mal gab es keinen internen Übertrag von Bit 6 nach Bit 7, aber es ist ein externer Übertrag aufgetreten. Das Ergebnis ist falsch, weil Bit 7 sich verändert hat. Es sollte eine Überlaufbedingung angezeigt werden.

Ein Überlauf tritt in folgenden vier Situationen auf:

1. bei der Addition großer positiver Zahlen,
2. bei der Addition betragsgroßer negativer Zahlen,
3. bei der Subtraktion einer großen positiven von einer betragsgroßen negativen Zahl,
4. bei der Subtraktion einer betragsgroßen negativen Zahl von einer großen positiven Zahl.

Lassen Sie uns nun unsere Definition des Überlaufs verbessern.

Technisch wird die Überlaufanzeige, ein besonders zu diesem Zweck reserviertes und *Kennzeichen (Flag)* genanntes Bit, gesetzt, wenn es einen Übertrag von Bit 6 nach Bit 7 und keinen externen Übertrag gibt. Es wird ebenfalls gesetzt, wenn es keinen Übertrag von Bit 6 nach Bit 7, aber einen externen Übertrag gibt. Das gibt an, daß sich Bit 7, das Vorzeichen des Ergebnisses, versehentlich geändert hat.

Das Überlaufkennzeichen wird durch die Verknüpfung des Überlaufeingangs und des Überlaufausgangs von Bit 7 (das Vorzeichenbit) durch ein exklusives Oder gesetzt; diese Anmerkung für den technisch interessierten Leser. Praktisch ist jeder Mikroprozessor mit einem besonderen Überlaufkennzeichen ausgestattet, um diese Bedingung automatisch zu prüfen – eine Bedingung, die korrigierendes Handeln erfordert.

Ein Überlauf zeigt an, daß das Ergebnis einer Addition oder Subtraktion mehr Bits erfordert als in dem Standard-8-Bit-Register, das das Ergebnis enthalten soll, zur Verfügung stehen.

Der Übertrag und der Überlauf

Das Übertrags- und das Überlaufsbit heißen *Kennzeichen (Flags)*. Sie sind in jedem Mikroprozessor vorgesehen. In Kapitel 2 werden wir lernen, sie zum effizienten Programmieren zu benutzen. Diese beiden Anzeigen befinden sich in einem besonderen Register, Kennzeichen- oder „Status"-Register genannt. Dieses Register enthält noch weitere Anzeigen (wie in Kapitel 4 beschrieben).

Beispiele Wir werden uns nun aktuelle Beispiele ansehen, die die Arbeitsweise des Übertrags und des Überlaufs veranschaulichen. In

jedem Beispiel bezeichnet das Symbol V den Überlauf und C den Übertrag. Wenn es keinen Überlauf gab, ist V gleich 0, anderenfalls ist V gleich 1 (dasselbe gilt für den Übertrag C). Erinnern Sie sich, daß die Regeln für das Zweier-Komplement besagen, daß der Überlauf nicht beachtet wird. (Der mathematische Beweis wird hier nicht angegeben.) Lassen Sie uns die folgenden Beispiele untersuchen:

Positiv-Positiv

binär	dezimal
00000110	6
+00001000	+8
00001110	14

(RICHTIG)
V = 0 , C = 0

Positiv-Positiv mit Überlauf

binär	dezimal
01111111	127
+00000001	+ 8
10000000	128

(FALSCH, da 10000000 der Dezimalzahl −128 entspricht.)
V = 1 , C = 0

Das obige Ergebnis ist falsch, weil ein Überlauf aufgetreten ist.

Positiv-Negativ (positives Ergebnis)

binär	dezimal
00000100	4
+11111110	+(−2)
(1)00000010	2

(RICHTIG)
V = 0 , C = 1 (zu vernachlässigen)

Positiv-Negativ (negatives Ergebnis)

binär	dezimal
00000010	2
+11111100	+(−4)
11111110	−2

(RICHTIG)
V = 0 , C = 0

Negativ-Negativ

binär	dezimal
11111110	−2
+11111100	+(−4)
(1)11111010	−6

(RICHTIG)
V = 0 , C = 1 (zu vernachlässigen)

Negativ-Negativ mit Überlauf

binär	dezimal
10000001	−127
+11000010	+(− 62)
(1)01000011	−189

(FALSCH, da 01000011 der Dezimalzahl +67 entspricht.)
V = 1 , C = 1

In dem letzten Beispiel ist durch die Addition zweier betragsgroßer negativer Zahlen ein „Unterlauf" aufgetreten. Das Ergebnis −189 ist betragsmäßig zu groß, um mit Hilfe von 8 Bits dargestellt zu werden.

Festformatdarstellung

Wir wissen nun, wie ganze Zahlen mit Vorzeichen darzustellen sind; wir haben jedoch noch nicht das Größenordnungsproblem gelöst. Wenn wir größere ganze Zahlen darstellen, benötigen wir mehrere Bytes. Um arithmetische Operationen effizient auszuführen, ist es besser eine feste, als eine variable Anzahl von Bytes zu benutzen. Deshalb steht die größte Zahl, die dargestellt werden kann fest, sobald die Anzahl der Bytes gewählt ist.

Das Größenordnungsproblem

Lassen Sie uns folgenden wichtigen Punkt betrachten: Die für die Zweier-Komplement-Darstellung gewählte Anzahl n von Bits liegt üblicherweise für das Programm fest. Wenn irgendein Ergebnis oder eine Zwischenrechnung eine Zahl erzeugen sollte, die mehr als n Bits erfordert, gehen einige Bits verloren. Das Programm behält normalerweise die am weitesten links stehenden (die signifikantesten) Bits und läßt die mit niedriger Ordnung fallen. Das nennt man „das Ergebnis *abschneiden*".

Wir wollen ein Beispiel im Dezimalsystem mit einer sechsziffrigen Darstellung betrachten:

```
   123456
x     1.2
---------
   246912

  123456
---------
  148147.2
```

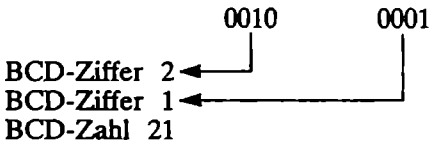
Das Ergebnis erfordert sieben Ziffern. Die 2 hinter dem Dezimalpunkt läßt man fallen, und das Endergebnis ist 148147. Es ist abgeschnitten worden. Solange die Position des Dezimalpunktes nicht verlorengeht, wird diese Methode üblicherweise benutzt, um den Zahlenbereich auf Kosten der Genauigkeit zu erweitern. Die Problematik ist in der binären Darstellung die gleiche. Diese Festformatdarstellung kann einen Verlust an Genauigkeit verursachen, aber sie ist für übliche Rechnungen oder mathematische Operationen ausreichend.

Leider kann man bei Buchhaltungsproblemen keinen Verlust an Genauigkeit hinnehmen. Wenn zum Beispiel ein Kunde eine höhere Summe zu bezahlen hätte, wäre es nicht akzeptabel, wenn die Registrierkasse den zu zahlenden Betrag durch eine fünfziffrige Darstellung annäherte. Also muß immer, wenn die Genauigkeit des Ergebnisses wesentlich ist, eine andere Darstellung benutzt werden. Die normalerweise gewählte Lösung ist *BCD* (Binary Coded Decimal) oder binär-codierte Dezimalziffer.

BCD-Darstellung

Das zur Darstellung der Zahlen im BCD-Code benutzte Prinzip besteht darin, jede einzelne Dezimalziffer getrennt zu codieren und so viele Bits zu benutzen, wie nötig sind, um die komplette Zahl exakt darzustellen. Um jede Ziffer von 0 bis 9 zu codieren, sind vier Bits nötig. (Drei Bits reichen nur für acht Kombinationen und genügen deshalb nicht, um zehn Ziffern zu codieren.) Vier Bits erlauben sechzehn Kombinationen und sind daher ausreichend. Es sollte auch beachtet werden, daß sechs der möglichen Kombinationen in der BCD-Darstellung nicht benutzt werden (siehe Abb. 1.4). Dies wird später bei der Ausführung von Additionen und Subtraktionen zu einem möglichen Problem werden. Da nur vier Bits gebraucht werden, um eine BCD-Ziffer zu codieren, können zwei BCD-Ziffern in jedem Byte untergebracht werden. Das nennt man *BCD-gepackt*. Zum Beispiel bedeutet 00000000 in BCD 00, 10011001 entspricht 99.

Ein BCD-Code wird folgendermaßen gelesen:

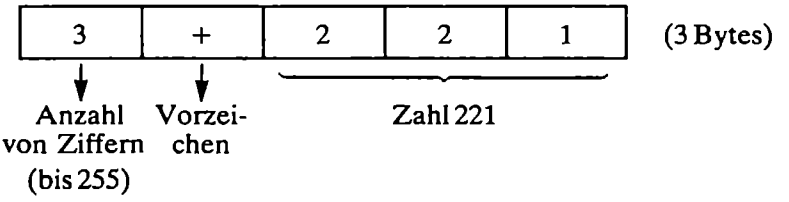


CODE	BCD-SYMBOL	CODE	BCD-SYMBOL
0000	0	1000	8
0001	1	1001	9
0010	2	1010	UNUSED
0011	3	1011	UNUSED
0100	4	1100	UNUSED
0101	5	1101	UNUSED
0110	6	1110	UNUSED
0111	7	1111	UNUSED

Abb. 1.4: Eine BCD-Tabelle

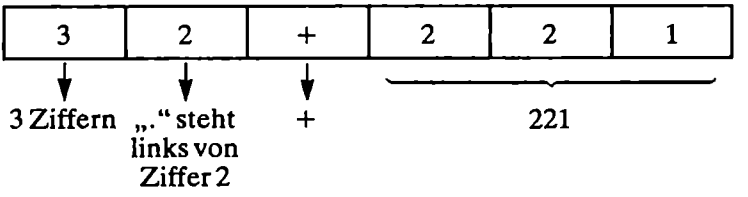
Es werden so viele Bytes benutzt, wie nötig sind, um alle BCD-Ziffern darzustellen. Typischerweise werden ein oder mehrere Nibbles am Anfang der Darstellung benutzt, um die Gesamtzahl der Nibbles (d.h. die Gesamtzahl der BCD-Ziffern) anzugeben. Ein weiteres Nibble oder auch ein Byte gibt die Position des Dezimalpunktes an. Die Konventionen können jedoch variieren.

Hier ist ein Beispiel für die Darstellung von ganzen Zahlen im BCD-Code mit mehreren Bytes:



Dieses Beispiel stellt +221 dar. (Das Vorzeichen kann zum Beispiel durch 0000 für + und 0001 für − dargestellt werden.)

Die BCD-Darstellung kann sich leicht den Dezimalzahlen anpassen. Zum Beispiel kann +2.21 durch



dargestellt werden.

Der Vorteil des BCD-Codes ist die Berechnung von absolut richtigen Ergebnissen. Sein Nachteil liegt darin, daß er einen großen Speicheraufwand und langsame arithmetische Operationen zur Folge hat. Dies ist nur im Zusammenhang mit Buchhaltungsproblemen hinnehmbar, aber der BCD-Code wird auch normalerweise in anderen Fällen nicht benutzt.

Wir haben nun Probleme gelöst, die mit der Darstellung positiver ganzer Zahlen, ganzer Zahlen mit beliebigen Vorzeichen und großer ganzer Zahlen zusammenhängen. Wir haben sogar eine mögliche Methode angegeben, Dezimalzahlen im BCD-Code darzustellen. Wir wollen nun das Problem angehen, Dezimalzahlen in einem festen Längenformat darzustellen.

Gleitpunktdarstellung

Das Grundprinzip der Gleitpunktdarstellung besteht darin, daß Dezimalzahlen in einem festen Längenformat dargestellt werden. Um keine Bits zu verschwenden, muß die Darstellung alle Zahlen *normalisieren.* 0.000123 verschwendet zum Beispiel drei Nullen links vor den von Null verschiedenen Ziffern. Diese Nullen haben außer der Angabe der Position des Dezimalpunktes keine Bedeutung. Die Normalisierung dieser Zahl führt zu $.123 \times 10^{-3}$. Dabei ist .123 die *normalisierte Mantisse;* -3 ist der *Exponent.* Wir haben diese Zahl normalisiert, indem wir alle bedeutungslosen Nullen links der ersten von Null verschiedenen Ziffer eliminiert und einen Exponenten hinzugefügt haben. Lassen Sie uns ein weiteres Beispiel betrachten.

Beispiel 1 22.1 wird zu $.221 \times 10^{2}$ normalisiert. Die allgemeine Form der Gleitpunktdarstellung ist $M \times 10^{E}$, wobei M die Mantisse und E der Exponent ist.

Wie man leicht sieht, läßt sich eine normalisierte Zahl dadurch charakterisieren, daß der Betrag ihrer Mantisse kleiner als 1 und größer oder gleich .1 ist, vorausgesetzt die Zahl ist von Null verschieden. Sie kann, mit anderen Worten, mathematisch dargestellt werden durch:

$$.1 \leq M < 1 \text{ oder } 10^{-1} \leq M < 10^{0}.$$

Ähnlich in der binären Darstellung:

$$2^{-1} \leq M < 2^{0} \text{ (oder } .5 \leq M < 1),$$

wobei M der absolute Wert der Mantisse (ohne Berücksichtigung des Vorzeichens) ist. Zum Beispiel:

111.01 wird normalisiert zu: $.11101 \times 2^{3}$.

Die Mantisse ist .11101. Der Exponent ist 3.

Nachdem wir nun das Prinzip der Darstellung definiert haben, lassen Sie uns das aktuelle Format untersuchen. Eine typische Gleitpunktdarstellung finden Sie in Abbildung 1.5.

In der Darstellung von Abb. 1.5 werden vier Bytes mit insgesamt 32 Bits benutzt. Das erste Byte links in der Abbildung wird zur Darstellung des Exponenten benutzt. Beide, Exponent und Mantisse, werden im Zweier-Komplement dargestellt. Das heißt der „größte" Exponent ist −128. „S" in Abb. 1.5 bezeichnet das Vorzeichenbit.

Drei Bytes werden zur Darstellung der Mantisse benutzt. Da das erste Bit in der Zweier-Komplement-Darstellung das Vorzeichen angibt, bleiben 23 Bits für die Darstellung der Größe der Mantisse.

Dies ist nur ein Beispiel für eine Gleitpunktdarstellung. Es ist möglich, nur drei Bytes zu benutzen, ebenso wie mehr Bytes zu verwenden. Die oben vorgeschlagene 4-Byte-Darstellung ist weitverbreitet, weil sie einen vernünftigen Kompromiß zwischen Genauigkeit, Größe der Zahlen, Speicherausnutzung und Effizienz der arithmetischen Operationen darstellt.

Wir haben nun die Probleme, die mit der Zahlendarstellung zusammenhängen, erforscht und haben gelernt, Zahlen in Form von ganzen Zahlen mit Vorzeichen oder in Dezimalform darzustellen. Lassen Sie uns nun mit der Untersuchung der internen Darstellung alphanumerischer Zeichen fortfahren.

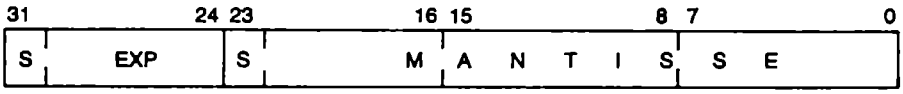


Abb. 1.5: Typische Gleitpunktdarstellung

Darstellung alphanumerischer Daten

Die Darstellung alphanumerischer Daten (d.h. Schriftzeichen) ist ganz einfach: alle Schriftzeichen werden in einem 8-Bit-Code verschlüsselt. In der Computerwelt sind nur zwei Codes im allgemeinen Gebrauch, der ASCII-Code und der EBCDIC-Code. ASCII steht für „American Standard Code for Information Interchange" (Amerikanischer Standardcode für Informationsaustausch) und ist universell verbreitet in der Welt der Mikroprozessoren. EBCDIC ist eine von IBM verwendete Abwandlung von ASCII und wird deshalb in der Mikrocomputerwelt nicht verwendet, außer man stellt eine Schnittstelle zu einem IBM-Terminal her.

Lassen Sie uns kurz die ASCII-Codierung untersuchen. Wir verschlüsseln 26 Buchstaben des Alphabets, große und kleine, plus 10 numerische Sym-

bole und vielleicht 20 zusätzliche Sonderzeichen. Das kann leicht mit 7 Bits, die 128 mögliche Kombinationen zulassen, erreicht werden (siehe Abbildung 1.6). Alle Schriftzeichen werden deshalb mit 7 Bits verschlüsselt.

Das achte Bit, wenn es verwendet wird, ist das *Paritätsbit.* Parität ist eine Technik, um zu prüfen, ob der Inhalt eines Bytes nicht versehentlich geändert worden ist. Die Anzahl der Einsen in dem Byte wird gezählt, und das achte Bit wird auf Eins gesetzt, wenn die Anzahl ungerade war, so daß die Gesamtzahl der Einsen gerade ist. Das nennt man *gerade Parität. Ungerade Parität* (d.h. das achte Bit [das äußerste linke Bit] so zu besetzen, daß die Gesamtzahl der Einsen in dem Byte ungerade ist) kann auch benutzt werden.

Als Beispiel lassen Sie uns das Paritätsbit für 0010011 berechnen, indem wir gerade Parität verwenden. Die Anzahl der Einsen ist 3. Das Paritätsbit muß deshalb 1 sein, so daß die Gesamtzahl der Einsen in dem Byte 4

HEX	MSD	0	1	2	3	4	5	6	7
LSD	BITS	000	001	010	011	100	101	110	111
0	0000	NUL	DLE	SPACE	0	@	P	_	p
1	0001	SOH	DC1	!	1	A	Q	a	q
2	0010	STX	DC2	"	2	B	R	b	r
3	0011	ETX	DC3	#	3	C	S	c	s
4	0100	EOT	DC4	$	4	D	T	d	t
5	0101	ENQ	NAK	%	5	E	U	e	u
6	0110	ACK	SYN	&	6	F	V	f	v
7	0111	BEL	ETB	'	7	G	W	g	w
8	1000	BS	CAN	(	8	H	X	h	x
9	1001	HT	EM	)	9	I	Y	i	y
A	1010	LF	SUB	*	:	J	Z	j	z
B	1011	VT	ESC	+	;	K	[	k	{
C	1100	FF	FS	,	<	L	\	l	|
D	1101	CR	GS	-	=	M	]	m	}
E	1110	SO	RS	.	>	N	^	n	~
F	1111	SI	US	/	?	O		o	DEL

Abb. 1.6: ASCII-Umwandlungstabelle (die Abkürzungen werden im Anhang B erläutert)

und damit gerade ist. Das Ergebnis ist 10010011, wobei die führende 1 das Paritätsbit ist und 0010011 das Schriftzeichen identifiziert.

Die 7-Bit ASCII-Code-Tafel zeigt Abb. 1.6. In der Praxis wird der ASCII-Code meist benutzt „wie er ist", ohne Paritätsbit, und das Byte wird durch Ergänzen einer 0 in der äußersten linken Position vervollständigt. Bei der Verwendung eines Paritätsbits wird dieses in geeigneter Form in der Bitposition 7 ergänzt.

In speziellen Situationen, wie im Fernmeldewesen, werden möglicherweise andere Codierungen, wie fehlerkorrigierende Codes, verwendet. Beschreibungen dieser Codes gehen jedoch über den Rahmen dieses Buches hinaus.

Nun, da wir die üblichen Darstellungen für beides, Programm und Daten, innerhalb des Computers betrachtet haben, lassen Sie uns die möglichen externen Darstellungen untersuchen.

Externe Darstellung von Information

Der Begriff „externe Darstellung von Information" bezieht sich auf die Art, wie Informationen dem *Benutzer,* d.h. im allgemeinen dem Programmierer, vorgestellt wird. Informationen können extern im wesentlichen in drei Formaten dargestellt werden: binär, oktal oder hexadezimal und symbolisch. Lassen Sie uns diese Formate untersuchen.

1. Binär Wir haben gesehen, daß Information intern in *Bytes,* Folgen von acht *Bits* (Nullen und Einsen), gespeichert wird. Es ist manchmal wünschenswert, diese interne Information direkt in ihrem binären Format – dies ist als *Binärdarstellung* bekannt – offenzulegen. Eine einfaches Beispiel liefern Leuchtdioden (LEDs, Light Emitting Diodes), die im wesentlichen Miniaturlämpchen auf der Frontplatte eines Mikrocomputers sind. Im Fall eines 8-Bit-Mikroprozessors ist eine Frontplatte typischerweise mit acht LEDs ausgestattet, um den Inhalt jedes internen Registers anzeigen zu können. Eine leuchtende LED bedeutet eine 1. Eine nichtleuchtende LED bedeutet eine 0. Solch eine binäre Darstellung kann für das genaue Austesten eines Programms verwendet werden, besonders wenn dieses Ein- und Ausgabe enthält, aber es ist natürlich auf menschlicher Ebene unpraktisch, weil es in den meisten Fällen einfacher ist, Informationen in symbolischer Form anzusehen. Zum Beispiel ist 9 viel einfacher zu verstehen und zu behalten als 1001. Geeignetere Darstellungen sind entwickelt worden, die die Schnittstelle zwischen Mensch und Maschinen verbessern.

2. Oktal und Hexadezimal Die oktale bzw. hexadezimale Darstellung verschlüsselt drei bzw. vier Bits in ein einziges Symbol. Das oktale Format

nimmt drei Bits, wobei jede Kombination von drei Bits durch ein Symbol zwischen 0 und 7 dargestellt wird (siehe Abb. 1.7). Ein Beispiel:

```
00  100  100  binär
 ↓    ↓    ↓
 0    4    4   oktal
```

Die Binärzahl 00100100 ergibt also oktal 044.

Ein weiteres Beispiel:

```
11  111  111  binär
 ↓    ↓    ↓
 3    7    7   oktal
```

Die Binärzahl 11 111 111 ergibt also Zahl 377.

Eine umgekehrte Umwandlung:

```
 2    1    1   oktal
 ↓    ↓    ↓
010  001  001  binär
```

Die Oktalzahl 211 ergibt also binär 10 001 001.

Die oktale Darstellung ist herkömmlicherweise in älteren Computern benutzt worden, die verschiedene Anzahlen von Bits, die von 8 bis viel-

BINÄR	OKTAL
000	0
001	1
010	2
011	3
100	4
101	5
110	6
111	7

Abb. 1.7: Oktale Symbole

leicht 64 reichen, benutzen. In neuerer Zeit ist mit der Vorherrschaft der 8-Bit-Mikroprozessoren das 8-Bit-Format zum Standard geworden, und es wird eine andere, praktischere Darstellung – die hexadezimale – verwendet.

In der hexadezimalen Darstellung wird eine Gruppe von vier Bits als eine hexadezimale Ziffer verschlüsselt. Hexadezimale Ziffern werden durch die Zahlen von 0 bis 9 und die Buchstaben A, B, C, D, E, F dargestellt. Zum Beispiel wird die Binärzahl 0000 durch hexadezimal 0 dargestellt; 0001 entspricht 1, und 1111 wird durch den Buchstaben F dargestellt (siehe Abb. 1.8).

Die Zahl $\underbrace{1010}_{A}\ \underbrace{0001}_{1}$ binär etwa wird durch

hexadezimal dargestellt.

Die hexadezimale Darstellung bietet den Vorteil, acht Bits in nur zwei Ziffern zu verschlüsseln. Dies ist leichter zu erfassen oder zu behalten und

DEZIMAL	BINÄR	OKTAL	HEX
0	0000	0	0
1	0001	1	1
2	0010	2	2
3	0011	3	3
4	0100	4	4
5	0101	5	5
6	0110	6	6
7	0111	7	7
8	1000	10	8
9	1001	11	9
10	1010	12	A
11	1011	13	B
12	1100	14	C
13	1101	15	D
14	1110	16	E
15	1111	17	F

Abb. 1.8: Hexadezimale Codes

schneller in einen Computer zu tippen als das zugehörige binäre Äquivalent. Deshalb wird auf den meisten neuen Mikrocomputern die hexadezimale Methode zur Darstellung von Bitgruppen bevorzugt.

Wenn die im Speicher vorhandene Information eine Bedeutung hat, wie zum Beispiel ein Text oder Zahlen, so ist die hexadezimale Form natürlich nicht geeignet, dem menschlichen Benutzer die Bedeutung dieser Information zu vermitteln.

3. Symbolische Darstellung *„symbolische Darstellung"* bezieht sich auf die externe Darstellung von Informationen in der tatsächlichen symbolischen Form. Dezimalzahlen werden zum Beispiel als Dezimalzahlen dargestellt und nicht als Folgen von hexadezimalen Symbolen oder Bits. Ähnlich wird Text als solcher dargestellt. Natürlich ist die symbolische Darstellung für den Benutzer am zweckmäßigsten. Sie wird verwendet, wenn eine geeignete Anzeigevorrichtung, wie eine CRT-Anzeige oder ein Drucker vorhanden ist. (Eine CRT-Anzeige ist ein dem Fernseher ähnlicher Bildschirm zur Darstellung von Text oder Grafiken.) Leider ist es unwirtschaftlich, in kleineren Systemen, wie Einplatinen-Mikrocomputern, solche Anzeigen vorzusehen, so daß der Benutzer dann auf die hexadezimale Kommunikation mit dem Computer beschränkt bleibt.

Zusammenfassung externer Darstellungen

Die symbolische Darstellung von Informationen ist die angenehmste, da sie für den menschlichen Benutzer die natürlichste ist. Sie erfordert jedoch eine teure Schnittstelle in Form einer alphanumerischen Tastatur plus einem Drucker oder einer CRT-Anzeige. Aus diesem Grund ist sie auf weniger teuren Systemen häufig nicht zu finden. Dann wird eine alternative Art der Darstellung benutzt, und in so einem Fall ist die hexadezimale Form vorherrschend. Nur in seltenen Fällen im Zusammenhang mit dem genauen Austesten auf der Hardware- oder Softwareebene wird die binäre Darstellung benutzt. Die *binäre Darstellung* zeigt die Inhalte der Register oder Speicher offen im binären Format an.

Nun, da wir gesehen haben, wie Information intern und extern dargestellt wird, lassen Sie uns mit der Untersuchung des eigentlichen Mikroprozessors, der diese Information verarbeitet, fortfahren.

Kapitel 2

Der innere Aufbau des 8086/8088

In diesem Kapitel werden wir den inneren Aufbau des 8086/8088-Mikroprozessors erörtern. Wir beginnen mit einer ausführlichen Diskussion über die Architektur eines Mikroprozessors im allgemeinen. Anschließend besprechen wir wichtige Konzepte, die den meisten Ein-Chip-Mikroprozessoren gemeinsam sind.

Als nächstes untersuchen wir den Befehlszyklus eines typischen Mikroprozessors. Am Ende dieser Diskussion sollen Sie ein gutes allgemeines Verständnis der internen Organisation eines Mikroprozessors und der Ausführung von Befehlen haben.

Wir fahren fort mit der Untersuchung der internen Organisation des 8086/8088. Wir schauen uns viele Einrichtungen, mit denen diese beiden Prozessoren ausgestattet sind, genau an und stellen sie den für andere Mikroprozessoren typischen Einrichtungen gegenüber. Beim Vergleich von Prozessoren kann man klar erkennen, daß die Konstrukteure des 8086/8088 für alte Probleme neue Lösungen gefunden haben und daß das zu einem leicht zu handhabenden, extrem leistungsfähigen Mikroprozessor geführt hat.

ALLGEMEINE MIKROPROZESSOR-SYSTEMARCHITEKTUR

Wir wollen unsere Ausführungen mit der Untersuchung der allgemeinen Hardwarearchitektur eines typischen auf einem Mikroprozessor basierenden Systems beginnen. Obwohl wir uns jetzt nicht die Zeit nehmen werden, zu erklären, wie diese Architektur mit dem 8086/8088 verwirklicht wird, werden wir zeigen, wie sich ein Mikroprozessor in eine Systemumgebung einfügt, und Ihnen dadurch helfen, die interne Organisation einer Zentraleinheit (CPU, central processing unit) besser zu verstehen.

Abb. 2.1 zeigt ein Blockdiagramm eines typischen von einem Mikroprozessor kontrollierten Systems. Links in dem Diagramm erscheint die *Mikroprozessoreinheit* (die *MPU*, microprocessor unit) – in diesem Fall der 8086/8088 – die die Funktionen der *Zentraleinheit* (*CPU*) auf einem

einzigen Chip vereinigt. Die CPU umfaßt eine *arithmetisch-logische Einheit* (die *ALU*, arithmetic-logical unit) zuzüglich ihrer internen Register und eine *Kontrolleinheit* (die CU, control unit), die Befehle entschlüsselt und intern aneinanderreiht. (Über die CPU lernen wir später in diesem Kapitel mehr.) Die MPU hat drei *Bussysteme*: einen *Adreß*bus, einen *Daten*bus und einen *Kontroll*bus. Ein Bus besteht aus einer Gruppe von Leistungen, die an einer gemeinsamen Quelle entspringen und eine gemeinsame Funktion ausführen.

In Abb. 2.1 sehen wir, daß neben der CPU ROM, RAM und E/A vorhanden sind. Ganz gleich, wie komplex ein System ist oder welches seine Funktion ist, das Blockdiagramm in Abbildung 2.1 beschreibt es genau.

Lassen Sie uns kurz jeden der Hauptblöcke, die mit ROM, RAM und E/A bezeichnet sind, untersuchen. Das *ROM* (oder read-only memory, nur lesbarer Speicher) speichert das Programm des Systems. Der Vorteil eines ROM-Speichers besteht darin, daß sein Inhalt beständig ist; er verschwindet nicht, wenn das System abgeschaltet wird. Aus diesem Grunde enthält das ROM üblicherweise das „Urlader-" oder Monitorprogramm, das es erlaubt, die einleitenden Systemoperationen durchzuführen. In

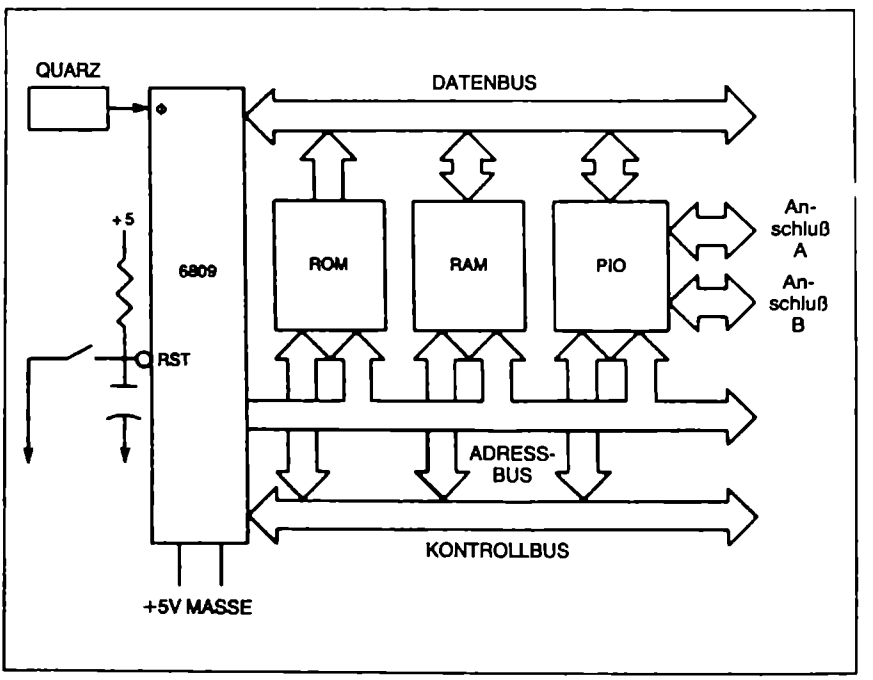


Abb. 2.1: Ein Blockschaltbild eines typischen Mikroprozessorsystems mit ROM, RAM und Ein-/Ausgabe

einer Prozeß-Kontrollanlage befinden sich fast alle Programme im ROM, weil sie selten verändert werden und gegen Stromausfälle geschützt werden müssen.

RAM (oder random access memory, Speicher mit wahlfreiem Zugriff) ist der Schreib-Lesespeicher des Systems. In einer Hobby- oder Programmentwicklungsanlage befinden sich die meisten Programme im RAM, so daß sie leicht verändert werden können. Solche Programme können im RAM behalten werden oder wenn das erwünscht ist, in das ROM übertragen werden. RAM ist flüchtig. Information im RAM gehen verloren, wenn der Strom abgeschaltet wird. In einem Kontrollsystem ist der Aufwand an RAM typischerweise klein (nur für Daten); in einem Programmentwicklungssystem jedoch ist der Aufwand an RAM im allgemeinen groß, da dort Programme zuzüglich Entwicklungssoftware gespeichert sind. Vor dem Gebrauch muß der Inhalt des RAM von einem externen Gerät geladen werden.

Schließlich enthält das System auch einen oder mehrere *Schnittstellenbausteine*. Ein Schnittstellenbaustein erlaubt die Kommunikation zwischen dem System und der externen Welt. Der PIO ist ein häufig verwendeter Schnittstellenbaustein. PIO steht für „parallel input/output" d.h. parallele Ein-/Ausgabe. PIOs sind wie die anderen Chips in dem System an alle drei Bussysteme angeschlossen und stellen Datenwege für die Kommunikation mit der Außenwelt her.

Lassen Sie uns nun die Funktion der Bussysteme studieren.

Die drei Bussysteme

Wie vorher erwähnt, hat das System in Abbildung 2.1 eine 3-Bus-Architektur. Der *Adreßbus* hat die Aufgabe, die Kommunikation zwischen der CPU und ROM, RAM oder E/A herzustellen. Der *Datenbus* überträgt die Information zwischen der CPU und dem aktivierten Systemblock (siehe Abbildung 2.1). Der *Kontrollbus* schließlich erfüllt die doppelte Aufgabe, erstens elektrisch den Typ der Kommunikation zu definieren und zweitens den Transfer zu beginnen und zu beenden.

In den meisten Anwendungen kontrolliert die CPU alle Busse. Es ist ihre Funktion, jeden Bus zeitlich so zu steuern, daß eine zuverlässige Kommunikation stattfinden kann.

Sie als Programmierer brauchen sich nicht mit dieser Tätigkeit, die von der Systemhardware ausgeführt wird, zu beschäftigen. Sie müssen der CPU jedoch die richtigen Befehle liefern, so daß sie die richtigen Adressen und Daten für das Hardwaresystem erzeugen kann. (Als Programmierer können Sie nur voraussetzen, daß die Hardware korrekt funktioniert.)

Die Breite eines Busses bestimmt die Anzahl von Signalleitungen, die den

Bus ausmachen. Für den 8086/8088 kann der Adreßbus bis zu 20 bit breit sein. Die Breite des Datenbusses ist 16 bit und die Breite des Kontrollbusses variiert, hat aber einen nominalen Wert von 5 bit. Die Breite jedes Busses variiert von CPU zu CPU und von System zu System.

DAS INNERE EINES MIKROPROZESSORS

Lassen Sie uns nun Abbildung 2.2 untersuchen und den allgemeinen internen Aufbau eines Mikroprozessors erörtern. Zusätzlich werden wir Ihnen zeigen, wie der 8086 und der 8088 ihre spezifische interne Architektur realisieren, und wir werden die Ähnlichkeiten und die Unterschiede zu dem allgemeinen (in Abbildung 2.2 gezeigten) Fall aufzeigen.

Abb. 2.2. zeigt ein Diagramm der Standardarchitektur eines üblichen Mikroprozessors. Wir wollen die verschiedenen Module in dieser Abbildung untersuchen. Der *Kontrollblock* (rechts in der Abbildung) stellt die Kontrolleinheit dar, die die Systemhardware synchronisiert und die zeitliche Koordination übernimmt.

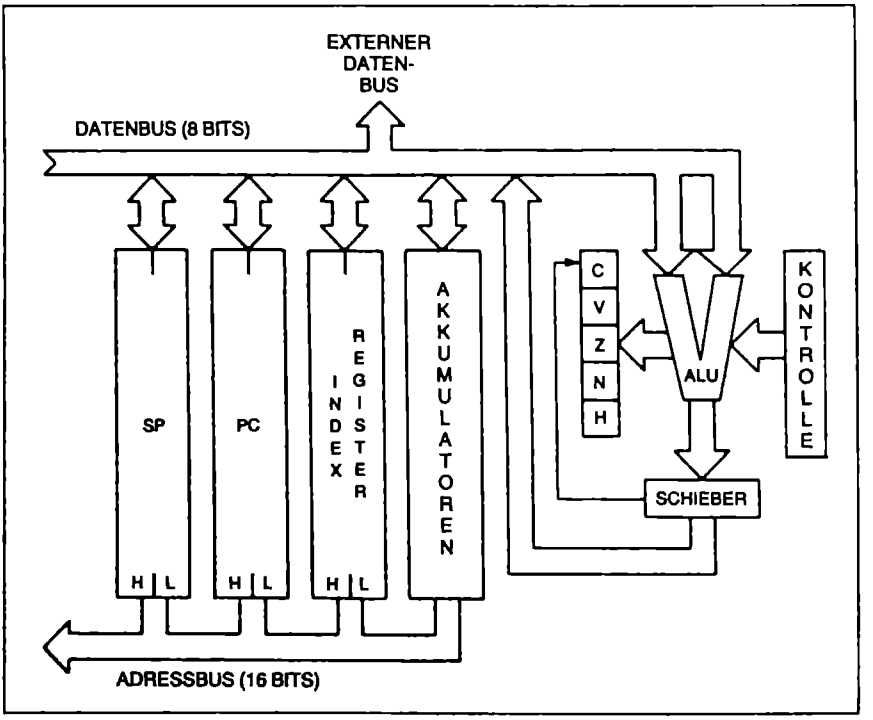


Abb. 2.2: Dieses Diagramm zeigt den allgemeinen internen Aufbau eines „Standard"-Mikroprozessors

Links von dem Kontrollblock ist die ALU. Die *ALU* führt alle arithmetischen und logischen Operationen aus. Spezielle Register, *Akkumulatoren* genannt, sind üblicherweise mit der Ausgabe der ALU verbunden. Sie enthalten die Ergebnisse arithmetischer Operationen. Zusätzlich zu den arithmetischen und logischen Operationen stellt die ALU auch *Verschiebe-* und *Rotations*einrichtungen zur Verfügung. Wie in Abbildung 2.3 dargestellt, bewegt eine Verschiebung den Inhalt des Akkumulators um eine oder mehrere Positionen nach rechts oder links. In dieser Abbildung ist jedes Bit um eine Position nach links bewegt worden.

Blicken wir zurück auf Abbildung 2.2, so erscheint das *Status- oder Zustandsregister* (condition flags) links von der ALU. Die Aufgabe dieses Registers ist es, die internen Zustände des Mikroprozessors in codierter Form zu speichern. Die Zustandsanzeigen werden manchmal auch *Bedingungskennzeichen* genannt. Ein Beispiel dafür ist ein Bit, das anzeigt, wenn das Ergebnis einer von der ALU ausgeführten Operation den Akkumulatorinhalt gleich 0 setzt. Wenn das auftritt, wird das Bedingungskennzeichen, das *Nullkennzeichen* (zero flag) genannt wird, auf wahr gesetzt.

Der Inhalt des Zustandsregisters kann durch besondere Befehle überprüft werden. Die CPU ändert ihren Ausführungsablauf aufgrund der

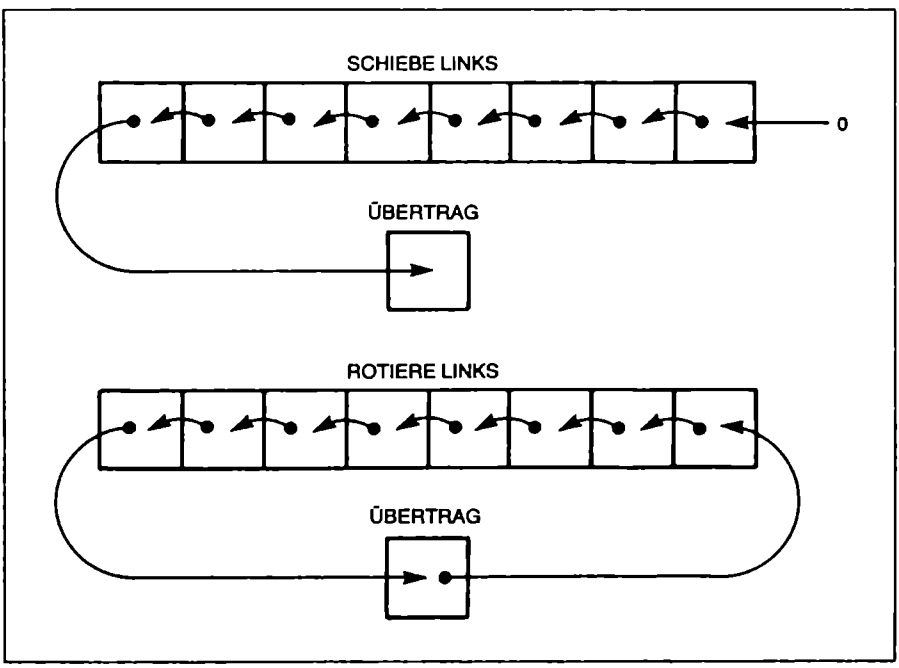


Abb. 2.3: Die ALU stellt auch Schiebe- und Rotationsmöglichkeiten zur Verfügung

logischen Werte dieser Flags. Wir werden dieses Thema detaillierter erörtern, wenn wir den Befehlssatz vorstellen und Programmbeispiele für den 8086/8088 geben. (Beachten Sie, daß die meisten Befehle, die vom Mikroprozessor ausgeführt werden, einige oder alle Bedingungskennzeichen verändern. Der Befehlssatz für den 8086/8088, der in Kapitel 4 vorgestellt wird, gibt genau an, welche Befehle welche Bedingungskennzeichen verändern.)

Bewegen wir uns in Abb. 2.2 nach links, finden wir die Adreßregister. Diese Register werden für die Speicherung von Adressen benutzt. Sie sind mit dem Systemadreßbus verbunden. Immer wenn die CPU ein Programm ausführt, werden die Adreßregister logisch verknüpft, um eine vollständige Systemadresse zu bilden. Ein Teil der Leistungsfähigkeit eines Mikroprozessors liegt in seiner Fähigkeit, Adreßregister auf besondere Arten zu verknüpfen, um die Systemadresse zu bilden, die auf dem Systembus anliegt. Kapitel 3 beschreibt im Detail, wie die Adreßregister

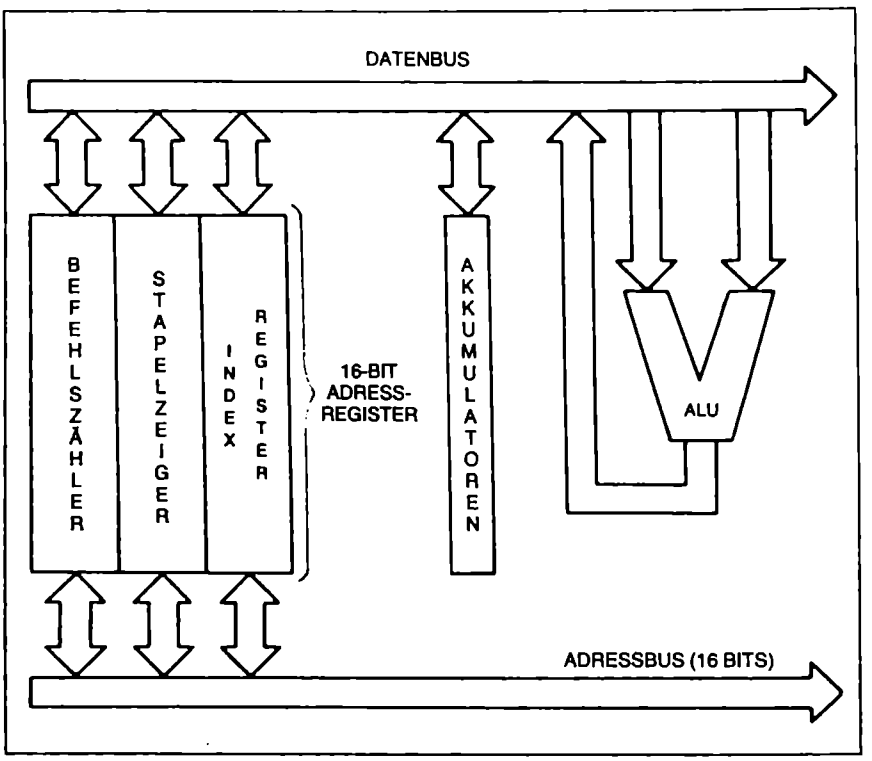


Abb. 2.4: Die PC-(program counter, Befehlszähler), SP- (stackpointer, Stapelzeiger) und Indexregister sind gebräuchliche Mikroprozessorregister zur Bildung einer vollständigen Systemadresse.

des 8086/8088 logisch verknüpft werden können, um eine Adresse zu bilden. Abb. 2.4 zeigt drei übliche in einem Mikroprozessor vorkommende Adreßregister: das Befehlszählerregister, den Stapelzeiger und das Indexregister. Beachten Sie, daß die Register auch in Abbildung 2.2 erscheinen. Wir wollen sie nun besprechen.

Das Befehlszählerregister (oder PC, program counter)

Das *Befehlszählerregister* muß in allen Mikroprozessoren vorhanden sein, da es für die Programmausführung unerläßlich ist. Es enthält die Adresse des nächsten auszuführenden Befehls.

Ein Programm läuft normalerweise sequentiell ab. Um den jeweils nächsten Befehl auszuführen, ist es notwendig, ihn aus dem Systemspeicher zu „holen" und in den Mikroprozessor zu laden. Der Inhalt des PC liegt dabei auf dem Adreßbus an und bestimmt die Adresse des jeweils nächsten auszuführenden Befehls. Der Speicher liest dann den durch die Adresse spezifizierten Inhalt und schickt den entsprechenden Befehl zurück an die CPU.

Der Stapelzeiger (oder SP, stack pointer)

Der *Stapelzeiger* verwaltet den Systemstapel. (Der Stapel wird in einem noch folgenden Abschnitt detailliert beschrieben. Er wird oft zur Behandlung von Unterbrechungen und Unterprogrammen und zur Rettung von temporären Daten benutzt.) Der Stapelzeiger enthält die Adresse des obersten Stapelelements.

Das Indexregister

Indizierung ist eine Speicheradressierungsmöglichkeit, die verwendet wird, um mit nur einem einzigen Befehl auf einen Block von Daten im Speicher zuzugreifen. Diese Einrichtung ist nicht in allen Mikroprozessoren vorhanden. Sie ist jedoch im 8086/8088 verfügbar. Ein *Indexregister* enthält typischerweise einen Verschiebeanteil, der bei der Berechnung der Adresse automatisch zum Basiswert addiert wird. Kurz, Indizierung wird beim Zugriff auf ein Wort in einem Datenblock benutzt. Nun, da wir alle Elemente in Abbildung 2.2 besprochen haben, lassen Sie uns fortfahren und den Stapel beschreiben.

Der Stapel

Ein Stapel (stack), formal eine LIFO-Struktur (last in – first out, zuletzt hinein – zuerst heraus) genannt, ist eine Menge von Speicherplätzen, die der Stapeldatenstruktur zugewiesen ist. Das wesentliche Merkmal eines Stapels ist, daß das erste in den Stapel eingelagerte Element immer unten und das zuletzt deponierte Element immer oben ist.

Ein Analogon kann man aus Tellern, die auf einem Restauranttresen gestapelt sind, ableiten, indem man annimmt, daß eine Öffnung im Tresen ist (mit einer Feder am Boden) und daß die Teller in dem Loch gestapelt sind. Bei dieser Anordnung ist gesichert, daß der Teller, der zuerst auf den Stapel gepackt wurde, immer unten ist und daß der zuletzt deponierte immer oben ist.

Normalerweise kann man auf einen Stapel nur mit zwei Befehlen zugreifen: PUSH und POP. Diese beiden Befehle werden in Abb. 2.5 veranschaulicht. Die PUSH-Operation legt Elemente oben auf dem Stapel ab. Ein POP-Befehl befördert das oberste Element des Stapels in das interne Register, das im Befehl spezifiziert wird.

DER ALLGEMEINE BEFEHLSZYKLUS

Lassen Sie uns nun Abbildung 2.6 untersuchen. Dieses Diagramm veranschaulicht die Rolle des Befehlszählerregisters, das einen Befehl aus dem Speicher holt. Die Mikroprozessoreinheit erscheint links in der Abbildung und der Speicher rechts. Der Speicher nimmt Befehle und Daten auf. Der Speicherbaustein kann entweder ein ROM, RAM oder irgendein anderer Baustein sein, der zufällig einen Speicher enthält.

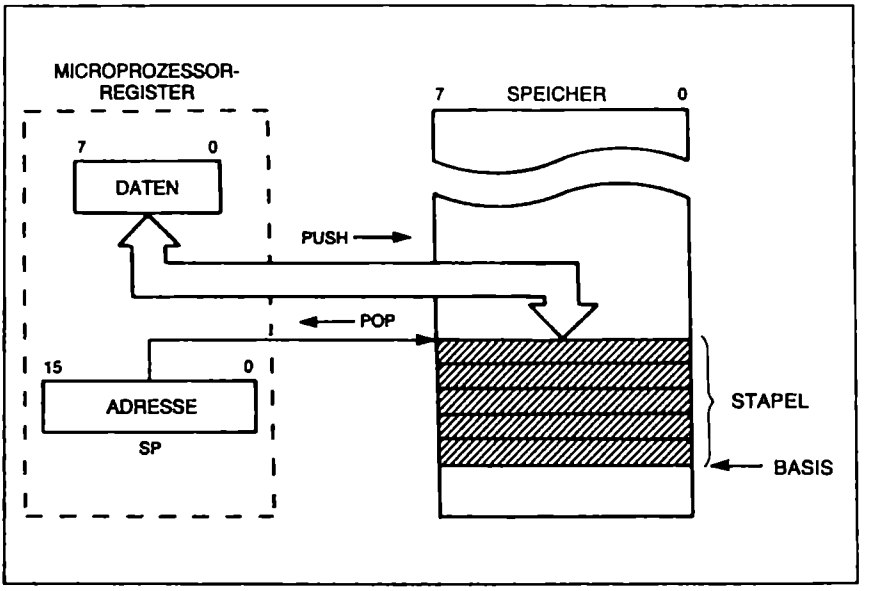


Abb. 2.5: Dieses Diagramm zeigt die beiden Befehle zur Stapelverwaltung: PUSH und POP

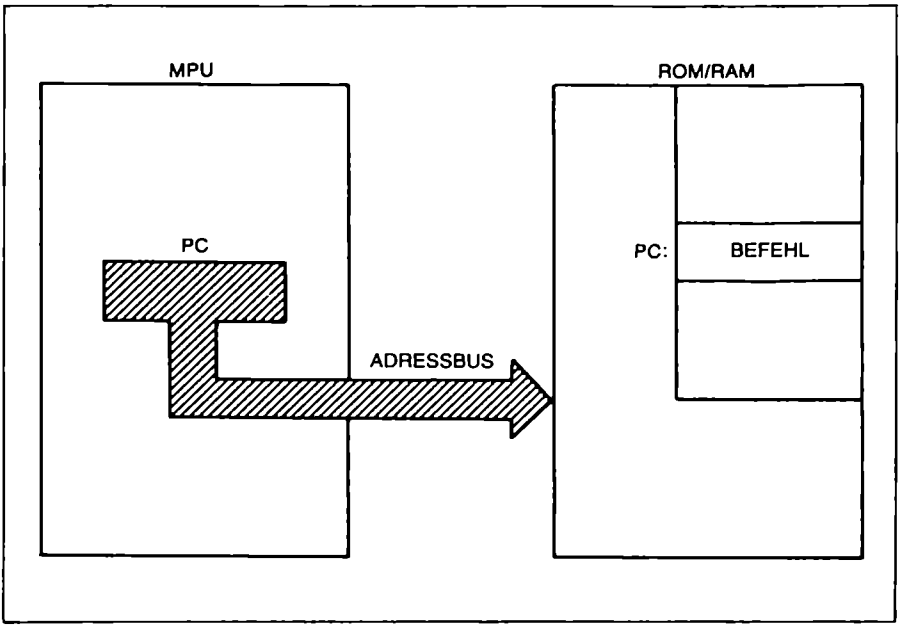


Abb. 2.6: Dieses Diagramm zeigt das Holen eines Befehls aus dem Speicher

Bei der Betrachtung von Abbildung 2.6 nehmen wir an, daß das Befehlszählerregister eine Adresse enthält. Das ist die Adresse des nächsten Befehls, der aus dem Speicher geholt werden soll.

Jeder Mikroprozessor geht in den folgenden drei Zyklen vor:

1. Holen des nächsten Befehls
2. Entschlüsseln des Befehls
3. Ausführen des Befehls

Lassen Sie uns nun diese Zyklen besprechen.

Das Holen

Im ersten Zyklus wird der Inhalt des Befehlszählerregisters auf den Adreßbus gebracht. Der Kontrollbus erzeugt dann ein Speicherlesesignal. Wenn das Lesesignal den Speicher erreicht, werden die Daten, die sich im Speicher an der angegebenen Adresse befinden, auf den Systemdatenbus gebracht. Der Mikroprozessor liest dann die Information auf dem Datenbus in ein internes Register, *Befehlsregister* (oder IR, instruction register), genannt. Die in die CPU gelesene Information ist der

Befehl. Wir können sagen, daß der Befehl aus dem Speicher geholt worden ist. Abbildung 2.7 ist ein Blockdiagramm, das das Holen eines Befehls aus dem Speicher zeigt.

Das Entschlüsseln und Ausführen

Sobald der Befehl im IR ist, entschlüsselt ihn die Kontrolleinheit des Mikroprozessors und erzeugt die richtige Folge von internen und externen Signalen für seine Ausführung. Die CPU braucht etwas Zeit, typischerweise einen Zeittakt, um einen Befehl zu entschlüsseln und logisch zu entscheiden, welche Handlung vorzunehmen ist. Wir werden später sehen, daß der 8086/8088 diese normalerweise „untätig verbrachte" Zeit (d.h. die Wartezeit auf die CPU, daß diese den Befehl entschlüsselt) zu einem vernünftigen Zweck verwendet.

Sobald die CPU den Befehl entschlüsselt hat, führt sie eine Reihe von Handlungen aus, die der Befehl vorschreibt. Einige Befehle erfüllen ein-

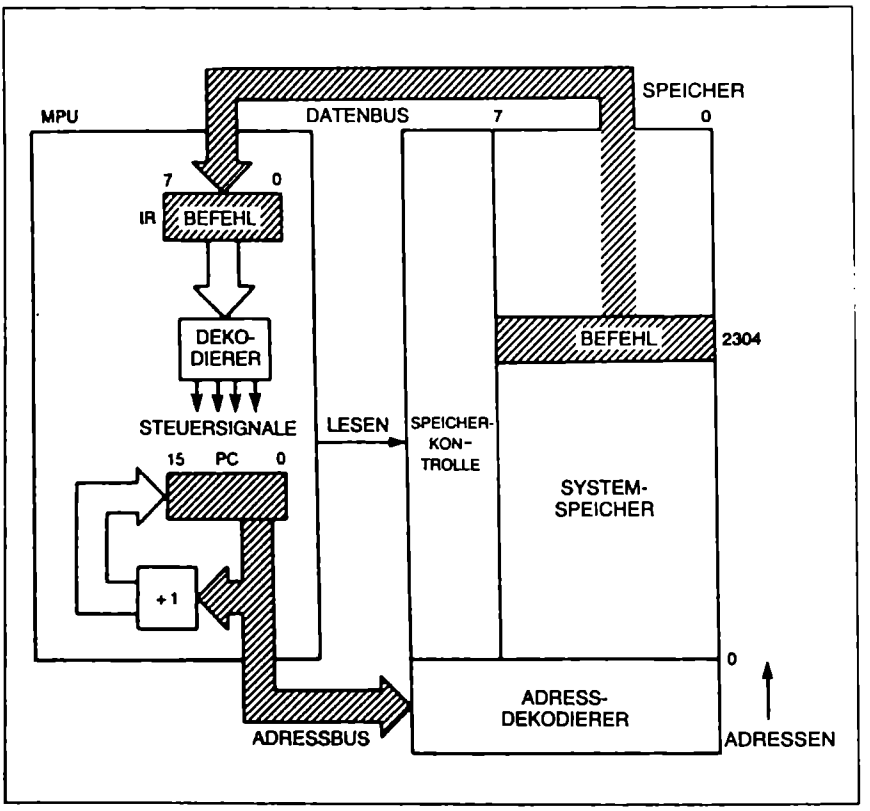


Abb. 2.7: Dieses Diagramm zeigt das Holen eines Befehls aus dem Speicher

fache Aufgaben und benötigen daher eine kurze Ausführungszeit; andere fordern das Eintreten mehrerer Ereignisse und dauern daher länger. Die Ausführungsgeschwindigkeit eines Befehls wird durch die Gesamtzahl der Zeittakte, die für die Erledigung erforderlich sind, bestimmt.

Das Holen des nächsten Befehls

Wir haben gerade beschrieben, wie ein Befehl unter Benutzung des Befehlszählerregisters aus dem Speicher geholt wird. Während der Ausführung eines Programms werden die Befehle nacheinander geholt. Deshalb ist ein automatischer Mechanismus erforderlich, *Inkrementierer* genannt, der dafür vorgesehen ist, die Befehle nacheinander zu holen. Der Inkrementierer ist an das Befehlszählerregister angeschlossen, wie es in Abbildung 2.7 gezeigt wird.

Es muß betont werden, daß die vorangegangenen Beschreibungen vereinfacht sind. In Wirklichkeit können einige Befehle zwei, drei oder sogar sechs bytes lang sein, und in solchen Fällen muß der PC logisch entscheiden, wie viele Bytes in jedem Befehl enthalten sind und wo der nächste Befehl beginnt. Wir werden mehr über die Funktion des PC diskutieren, wenn wir eingehend über die interne Architektur des 8086/8088 berichten.

DIE INTERNE ORGANISATION DES 8086/8088

Bis jetzt haben wir (in allgemeiner Form) erörtert, wie ein Mikroprozessorsystem aufgebaut ist, und wir haben die interne Architektur einer CPU untersucht. Nun wollen wir auf dieser Information aufbauen und über den tatsächlichen internen Aufbau des 8086/8088 eingehend berichten. Wenn Sie diesen Aufbau verstanden haben, ist es viel einfacher, die der Adressierungsarten (die im folgenden Kapitel behandelt werden) zu verstehen.

Die interne Architektur des 8086 und die des 8088 sind unter dem Gesichtspunkt der Programmierung eigentlich identisch. Software, die für den 8086 geschrieben wurde, läuft ohne Änderungen auf dem 8088. Die Umkehrung ist ebenfalls richtig. *(Beachten Sie:* Es ist wichtig, zu bemerken, daß der Hardwareaufbau jedes Mikroprozessors ganz einzigartig ist. Das ist für uns hier jedoch von geringem Interesse, da wir uns in diesem Buch auf die Programmierung konzentrieren.)

Abb. 2.8 zeigt ein Blockdiagramm der Funktion des 8086/8088. Diese Abbildung zeigt, daß es zwei große funktionale logische Blöcke in dem Baustein gibt: die BIU (bus interface unit oder Busschnittstelleneinheit) und die EU (execution unit oder Ausführungseinheit). In dem allgemeinen Blockdiagramm oben in Abb. 2.2 waren diese beiden Blöcke in einem einzigen Kontrollblock zusammengefaßt. Wir wollen untersuchen, was sie leisten.

Die Hauptfunktion der BIU besteht darin, die physikalische Schnittstelle zwischen 8086/8088 und der Außenwelt zur Verfügung zu stellen. Die BIU kontrolliert Adreß, Daten- und Kontrollbus des Systems. Sie kann parallel zur EU arbeiten. Eine der einzigartigen Einrichtungen der BIU ist ihre „Befehls-Vorausschau"-Fähigkeit. Um zu verstehen, was dies bedeutet, betrachten Sie folgendes.

Wenn die CPU Befehle ausführt, holt sie zuerst den Befehl aus dem Speicher und entschlüsselt ihn anschließend. Während der Entschlüsselungszeit sind die externen Busse der CPU untätig. Mit anderen Worten: Es gibt keine Aktivität auf den Bussen, weil die CPU nicht weiß, was zu tun ist, da der Befehl noch nicht entschlüsselt worden ist.

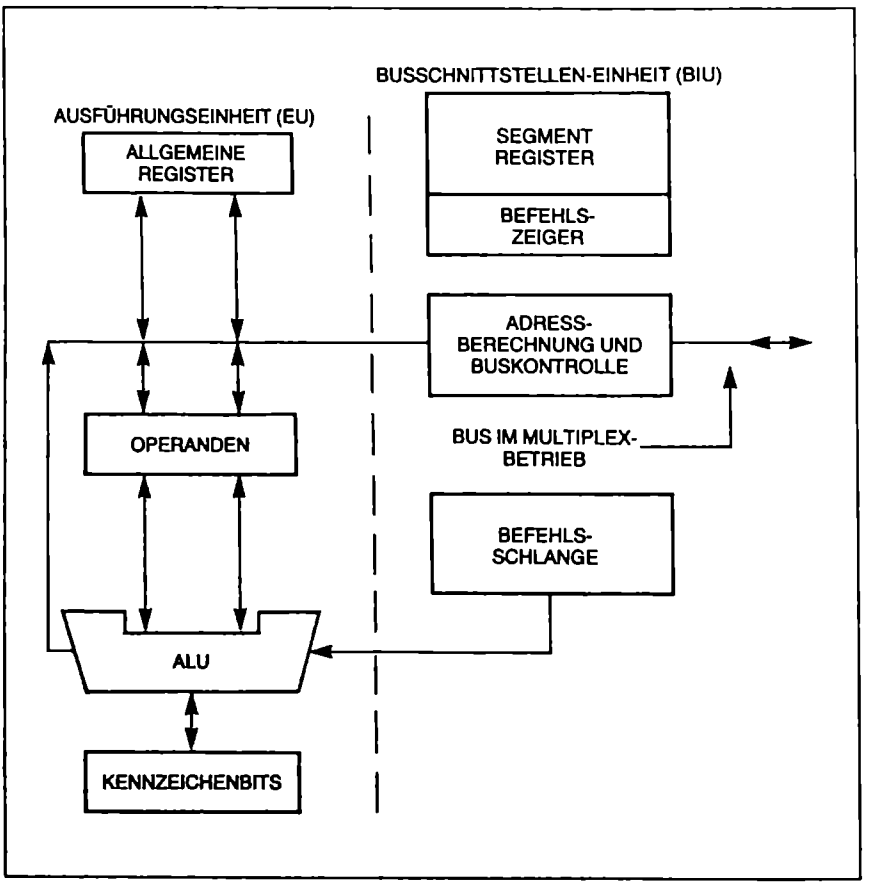


Abb. 2.8: Dieses Blockdiagramm zeigt die beiden grundlegenden funktionalen Blöcke innerhalb des 8086/8088. Diese sind die EU (execution unit, Ausführungseinheit) und die BIU (bus interface unit, Busschnittstelleneinheit)

Im 8086/8088 holt die BIU einen Befehl aus dem Speicher, und während die EU diesen Befehl noch decodiert, holt die BIU schon den nächsten Befehl herein.

Auf diese Weise wird also der nächste Befehl „vorausschauend hereingeholt" und er wartet dann auf die Ausführung durch die CPU. Tatsächlich kann der 8086 bis zu sechs Bytes Information in der Befehlsschlange haben, während der 8088 bis zu vier haben kann.

Einige Befehle auf dem 8086/8088 werden ganz intern in der CPU ausgeführt und können viele Zeittakte erfordern. Ein Beispiel für diesen Typ von Befehl ist der DIVIDE-Befehl. Während der Zeit der Division in der CPU holt die BIU „vorausschauend" Daten aus dem Speicher, bis zur maximalen Anzahl von Bytes, die die Schlange aufnehmen kann.

Daten aus dem Speicher vorausschauend hereinzuholen verkürzt die gesamte Ausführungszeit der Befehle, denn die Zeit für das Holen des Befehls muß nicht in der Ausführungszeit berücksichtigt werden.

Diese Behauptung ist jedoch nicht immer richtig, weil es Fälle gibt, in denen ein ganzer Befehl ohne ein Vorhereinholen geladen werden muß. Dies tritt immer dann auf, wenn die Warteschlange geleert oder neu gefüllt wird. Eine solche Situation liegt z. B. vor, wenn der 8086/8088 einen Sprungbefehl ausführt. Wenn Sie nicht mit diesem Befehl vertraut sind, verstehen Sie möglicherweise nicht, warum die Schlange geleert werden muß. Dies wird erklärt, wenn der Befehlssatz in Kapitel 4 erläutert wird.

Der andere in Abbildung 2.8 gezeigte Hardwareabschnitt des 8086/8088 ist die Ausführungseinheit. Diese Einheit enthält eine 16-Bit arithmetisch-logische Einheit. Sie verwaltet die Bedingungskennzeichen der CPU. Die EU bearbeitet ebenfalls die internen Register der CPU. Alle Register in der EU sind 16 bit breit. Dies gilt für den 8086 und den 8088.

DIE ALLGEMEINEN REGISTER DES 8086/8088

Abb. 2.9 zeigt die allgemeinen Register des 8086/8088. In dieser Darstellung erkennen wir zwei Hauptgruppen: Die Datengruppe und die Zeiger- und Indexgruppe. Jedes Register ist 16 bit breit. Die Datengruppe ist mit AX, BX, CX, und DX bezeichnet. Sie können jeweils als ein 16-Bit-Register oder als zwei 8-Bit-Register benutzt werden. Wenn sie als zwei 8-Bit-Register benutzt werden, unterteilt man sie in eine obere (H) und eine untere (L) Hälfte und bezeichnet sie mit AH, AL, BH, BL, CH, CL und DH, DL.

Die Register der Zeiger- und Indexgruppe (dargestellt in Abbildung 2.9) werden nur als 16-Bit-Register benutzt. Diese Register werden mit SP (stack pointer oder Stapelzeiger), BP (base pointer oder Basiszeiger), SI

(source index oder Quellindex) und DI (destination index, oder Zielindex) bezeichnet. Die Gründe für diese Namen werden später klar werden, wenn wir die verschiedenen Adressierungsarten erörtern und Programmbeispiele für den 8086/8088 angeben. Im Augenblick ist es nur wichtig zu wissen, daß diese Register existieren und wie sie organisiert sind.

Segmentregister

Es gibt vier Segmentregister im 8086/8088, wie Abbildung 2.10 zeigt. Hier geben wir nur eine kurze Beschreibung der Segmentregister. Eine detaillierte Beschreibung folgt in Kapitel 3. Jedes Segmentregister ist 16 bit breit und jedes hat einen besonderen Namen: Codesegment (CS, code segment), Datensegment (DS, data segment), Stapelsegment (SS, stack segment) und Extrasegment (ES, extra segment). Die Namen helfen zu verstehen, wie die Segmentregister während der Programmausführung benutzt werden.

Diese Register werden mit anderen internen Registern während der Programmausführung kombiniert, um eine vollständige 20-Bit-Systemadresse für den Adreßbus außerhalb der CPU zu bilden. Bei der Ausfüh-

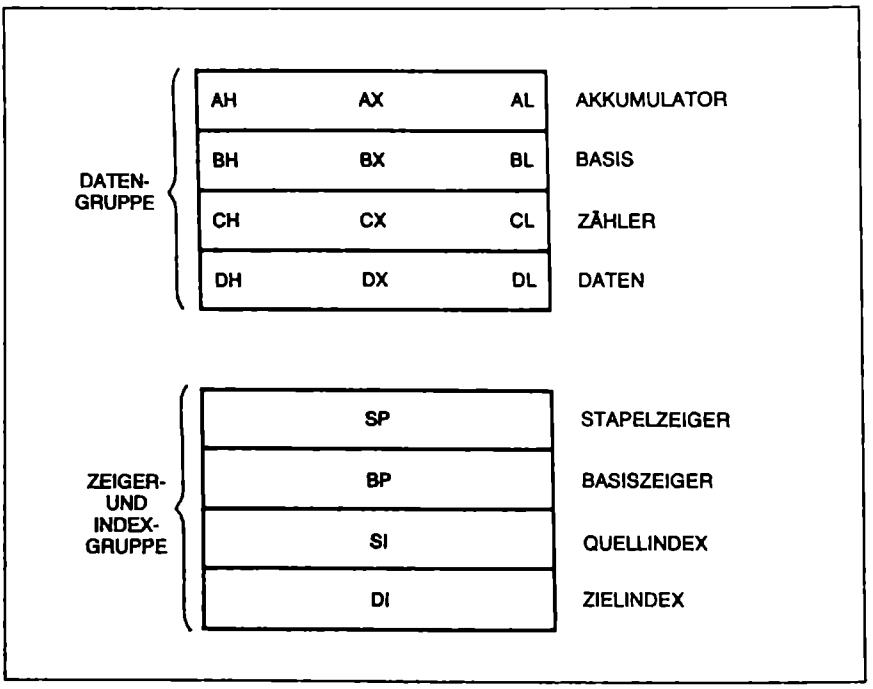


Abb. 2.9: Dieses Diagramm zeigt die allgemeinen internen Register des 8086/8088.

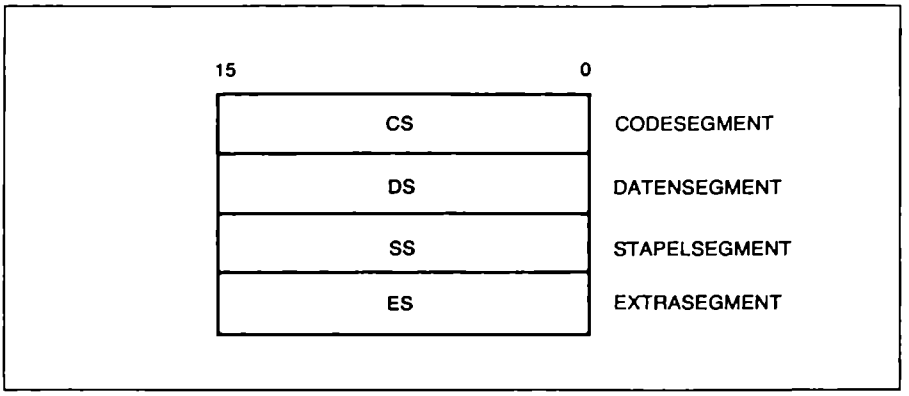


Abb. 2.10: Dieses Diagramm zeigt die vier Segmentregister des 8086/8088. Diese Register werden mit den allgemeinen Registern zur Bildung einer 20-Bit-Systemadresse kombiniert

rung einer PUSH- oder POP-Operation zum Beispiel wird das SS-Register mit anderen Registern logisch kombiniert, um die Speicheradresse, die auf das oberste Stapelelement zeigt, zu bilden. Wir werden dies eingehender in der Diskussion über die Adressierungsarten in Kapitel 3 erörtern.

Befehlszeiger

In der Beschreibung des allgemeinen Blockdiagramms eines Mikroprozessors, dargestellt in Abbildung 2.2, haben wir die Funktionsweise des Befehlszählerregisters behandelt. Im 8086/8088 wird das Befehlszählerregister durch den Befehlszeiger (IP, instruction pointer) ersetzt. Der IP wird durch die BIU aktualisiert, um auf die Adresse des nächsten auszuführenden Befehls zu zeigen. Programme haben keinen direkten Zugriff auf den Befehlszeiger; aber während der Ausführung eines Programms kann der IP verändert, auf den Stapel gerettet und vom Stapel wieder entnommen werden.

Kennzeichen

Der 8086/8088 hat sechs Status- oder Bedingungskennzeichen, die von der EU aktualisiert werden. Diese in Abbildung 2.11 gezeigten Kennzeichen stellen den Zustand des Ergebnisses einer gerade ausgeführten arithmetischen oder logischen Operation dar.

In Kapitel 4 werden Sie eine Gruppe von Befehlen finden, die dem Programm erlauben, seinen Ausführungsverlauf in Abhängigkeit vom logischen Wert dieser Kennzeichen zu ändern.

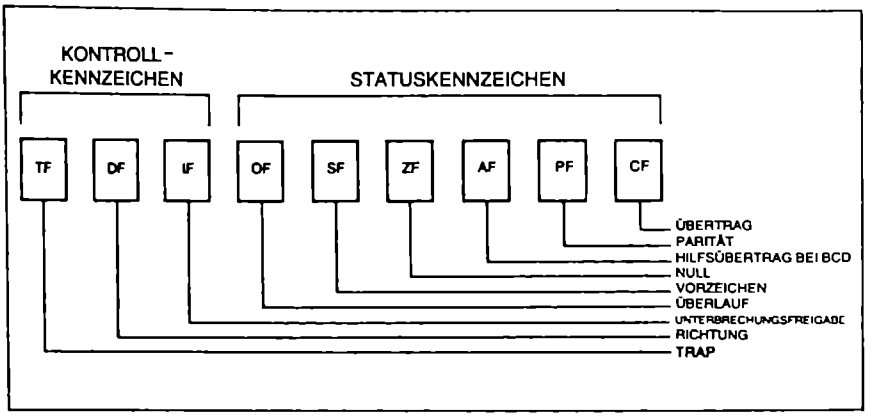


Abb. 2.11: 8086/8088 System-Kennzeichenbits

Kennzeichendefinitionen

AF *auxiliary carry flag* oder *Hilfsübertragskennzeichen*. Wenn dieses Kennzeichen gesetzt ist, hat es einen Übertrag vom unteren Nibble zum oberen Nibble oder aber ein Borgen vom oberen Nibble zum unteren gegeben. Das „obere und untere Nibble" bezieht sich auf das untere Byte eines 16-Bit Wertes.

CF *carry flag* oder *Übertragskennzeichen*. Dieses Kennzeichen ist gesetzt, falls es einen Übertrag oder ein Borgen bezüglich des höchstwertigen Bits des (8-Bit- oder 16-Bit-) Ergebnisses gegeben hat. (Wir haben das allgemeine Übertragskennzeichen in Kapitel 1 besprochen.)

OF *overflow flag* oder *Überlaufkennzeichen*. Wenn dieses Kennzeichen gesetzt worden ist, ist ein arithmetischer Überlauf aufgetreten. Dies bedeutet, daß die Größe des Ergebnisses die Speicherkapazität des Ziels überstieg und eine signifikante Stelle verlorengegangen ist. (Das allgemeine Überlaufkennzeichen wurde in Kapitel 1 behandelt.)

SF *sign flag* oder *Vorzeichenkennzeichen*. Dieses Kennzeichen wird gesetzt, wenn das höchste Bit des Ergebnisses eine logische 1 ist. Da negative binäre Zahlen mit Hilfe der Zweier-Komplement-Notation dargestellt werden, gibt SF das Vorzeichen des Ergebnisses an: eine 0 zeigt eine positive Zahl an und eine 1 eine negative.

PF *parity flag* oder *Paritätskennzeichen*. Wenn dieses Kennzeichen gesetzt ist, enthält das Ergebnis einer Operation eine gerade Anzahl von Einsen. Dieses Kennzeichen kann zur Aufdeckung von Datenübertragungsfehlern verwendet werden.

ZF *zero flag* oder *Nullkennzeichen*. Dieses Kennzeichen wird gesetzt, wenn das Ergebnis einer Operation Null ist.

Abbildung 2.12 zeigt alle internen Register des 8086/8088.

ZUSÄTZLICHE KONTROLLKENNZEICHEN

Es gibt drei Kontrollkennzeichen im 8086/8088, die wir noch nicht besprochen haben. Wir haben es vorgezogen, sie jetzt nicht zu behandeln, weil

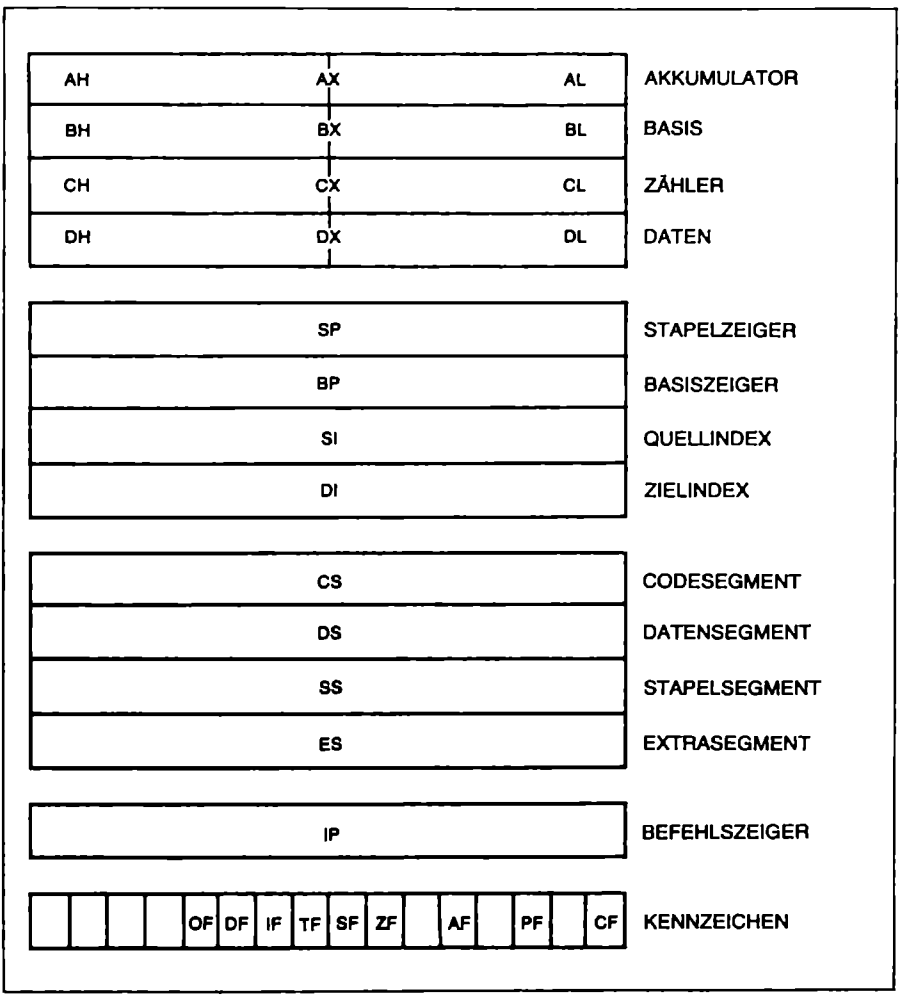


Abb. 2.12: Dieses Diagramm zeigt die internen Register des 8086/8088

es nützlicher sein wird, sie zu besprechen, wenn wir mehr über den 8086/8088 Befehlssatz wissen. Diese drei Kennzeichen werden mit DF (direction flag oder Richtungskennzeichen), IF (interrupt enable flag oder Unterbrechungs-Freigabe-Kennzeichen) und TF (trap flag oder Trapkennzeichen) bezeichnet.

ZUSAMMENFASSUNG

In diesem Kapitel haben wir den grundlegenden Aufbau eines Mikroprozessorsystems ausführlich behandelt. Wir haben ROM, RAM und E/A beschrieben und haben erklärt, wie sie über die Drei-Bus-Architektur mit dem Prozessor verbunden sind. Wir haben jeden Bus durch seine Funktion im System definiert. Zusätzlich haben wir das interne Blockdiagramm eines Mikroprozessors untersucht. Wir haben jeden größeren Block und sein Zusammenwirken mit den anderen Blöcken erforscht. Schließlich haben wir den inneren Aufbau des 8086/8088 besprochen, ausgehend von den Hauptblöcken der CPU – der BIU und EU –. Anschließend haben wir unsere Betrachtung stärker konzentriert, um die internen Register des 8086/8088 mit einzubeziehen.

Im nächsten Kapitel werden wir das Gelernte benutzen, um mehr über den 8086/8088 zu lernen. Insbesondere werden wir untersuchen, wie der 8086/8088 die internen Register gebraucht, um eine bestimmte Speicheradresse zu bilden.

Kapitel 3

Die Speicherorganisation und die Adressierungsarten des 8086/8088

In diesem Kapitel untersuchen wir die Systemspeicherorganisation der 8086- und 8088-Mikroprozessoren. Wir zeigen die zahlreichen Ähnlichkeiten und die Unterschiede zwischen den beiden Architekturen auf und erörtern für beide Prozessoren die verschiedenen Adressierungsarten. Wir betrachten auch Beispiele, die verdeutlichen, wie die einzelnen Adressierungsarten funktionieren. Schließlich lernen wir, wie Objekt- oder Binärcode für den 8086/8088 erzeugt wird und behandeln einige wichtige Punkte, die das Schreiben von Objektcode betreffen.

DIE SYSTEMSPEICHERORGANISATION DES 8086

Der 8086 hat 20 Adreßleitungen. Das bedeutet, daß 2^{20} oder 1 048 576 einzelne Speicherzellen adressiert werden können. Da der Datenbus des 8086 16 bit breit ist, ist der Systemspeicher für den 8086 ebenfalls 16 bit (oder zwei byte) breit. Diese beiden Bytes werden als oberes und unteres Byte bezeichnet, wie in Abbildung 3.1 dargestellt.

Der 8086 kann auf jedes der 1 048 576 möglichen Speicherbytes zugreifen. Da jedes Speicherwort zwei Bytes belegt, besteht der Systemspeicher aus 2^{19} Adressen zu je 16 Bits. (Es ist wichtig, sich zu erinnern, daß der 8086 sowohl auf Bytes als auch auf Wörter im Speicher zugreifen kann.)

Wie wir in einem späteren Kapitel sehen werden, setzen sich die Befehle des 8086 aus 1 bis 6 Datenbytes zusammen. Das bedeutet, daß die 8086-Befehle sowohl an einer geraden (unteres Byte) als auch an einer ungeraden (oberes Byte) Adresse beginnen können, wie Abbildung 3.2 zeigt. Zwei Signale bestimmen, auf welches Byte des Speichers zugegriffen wird: A0 und BHE.

Wenn A0 logisch 0 ist, greift der 8086 auf das untere Byte der gültigen Speicheradresse zu. Wenn BHE logisch 0 ist, greift der 8086 auf das obere Byte der gültigen Speicheradresse zu.

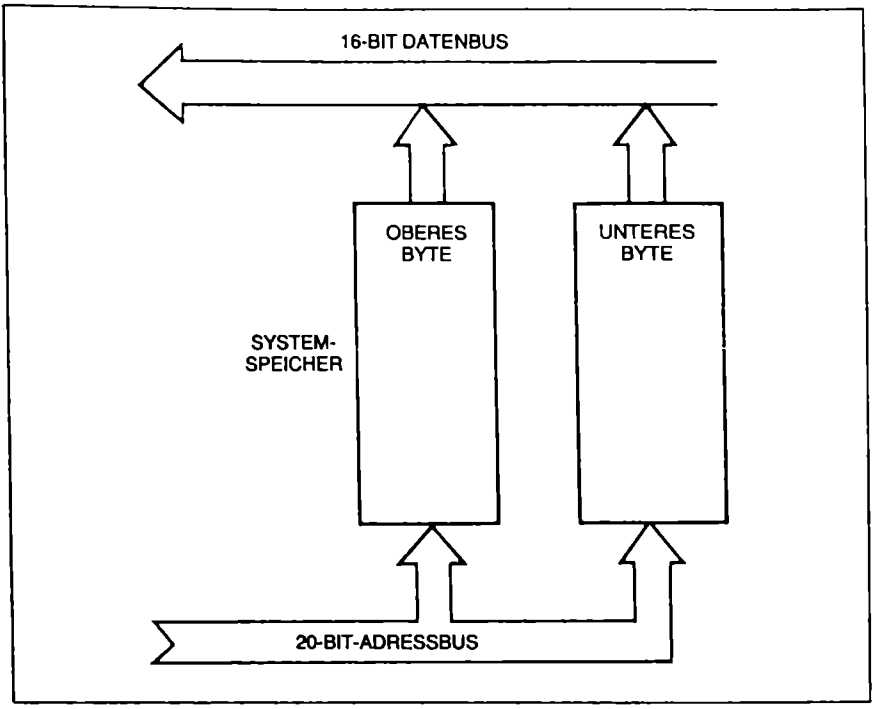


Abb. 3.1: Dieses Blockdiagramm zeigt, wie die 16 Bits des 8086-Speichers in obere und untere Bytes aufgeteilt werden

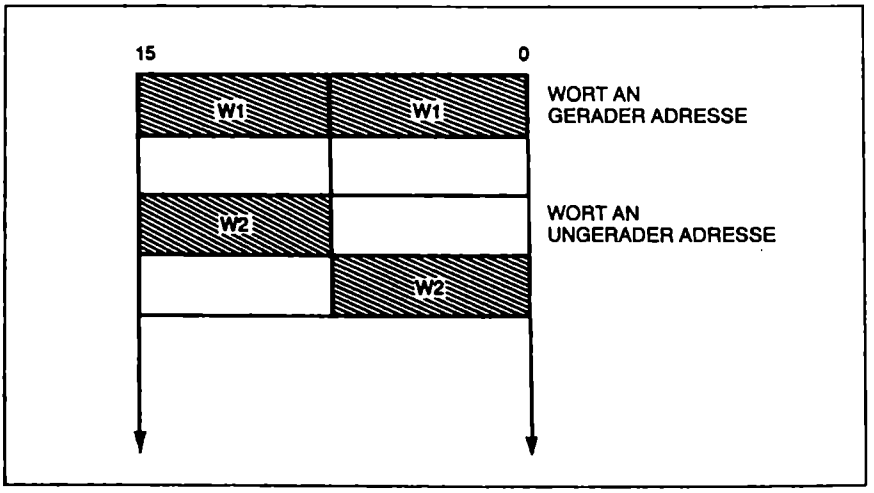


Abb. 3.2: Im Speicher des 8086 können Wörter (16 Bits) sowohl an gerader als auch an ungerader Adresse abgelegt werden

Der 8086 kann auf 16 Datenbits (oder zwei Bytes) des Speichers in einem einzigen Speicherzyklus zugreifen. Wenn das auftritt, sind BHE und A0 beide gleichzeitig logisch 0. Als Programmierer brauchen Sie diese Signale nicht zu kennen; es ist jedoch wichtig zu wissen, daß jedes Byte im Speicher getrennt adressiert werden kann, wenn bestimmte 8086-Befehle benutzt werden.

16-Bit-Zugriff an gerader Adresse

Um ausführlicher zu erklären, wie der 8086 auf Daten im Speicher zugreift, wollen wir einige Beispiele untersuchen. Lassen Sie uns annehmen, daß der 8086 ein 16-Bit-Wort von der Adresse 000F0, einer geraden Adresse, liest. Das bedeutet, daß der 8086 beide, das obere und das untere Datenbyte an der Adresse benötigt. Die CPU gibt dann die Adresse auf den Adreßbus aus, wobei BHE und A0 logisch 0 gesetzt sind. Beide Datenbytes liegen gleichzeitig (parallel) an der CPU an, wie in Abbildung 3.3 zu sehen ist.

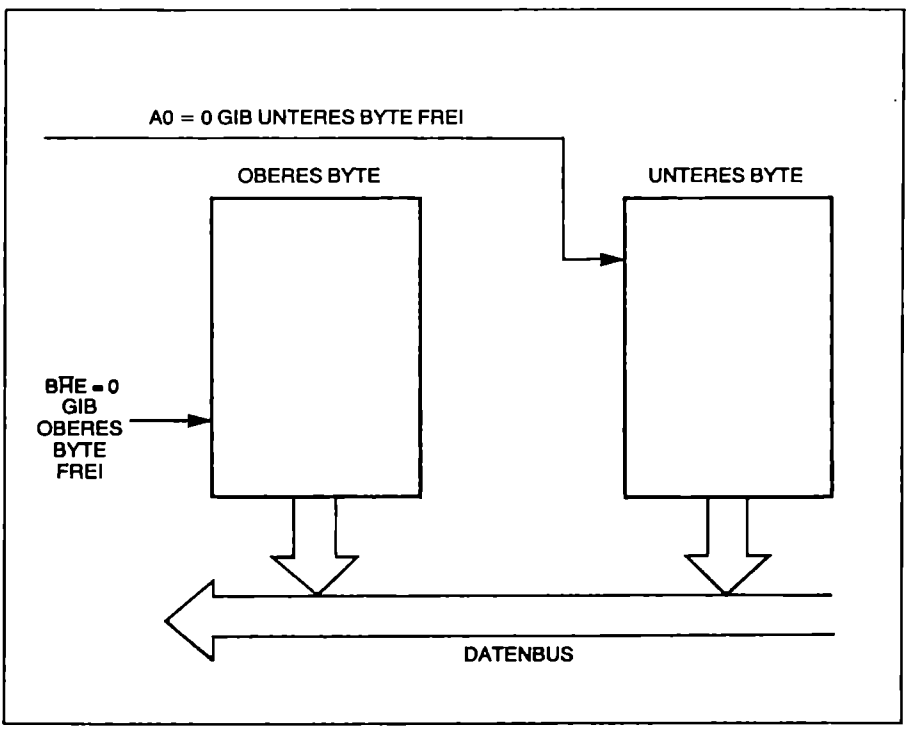


Abb. 3.3: Wenn der 8086 auf ein Wort zugreift, das an einer geraden Adresse im Systemspeicher liegt, werden beide, oberes und unteres Byte, auf dem Systembus freigegeben. In diesem Fall ist A0 gleich 0 und BHE gleich 0

8-Bit-Zugriff an gerader Adresse

Wir wollen nun annehmen, daß der 8086 ein Byte von der Speicheradresse 000F0 liest. Da diese Adresse gerade ist, werden die Daten des unteren Bytes an der Wortadresse gelesen. In diesem Fall hat A0 den Wert logisch 0 und BHE ist logisch 1.

8-Bit-Zugriff an ungerader Adresse

Lassen Sie uns nun annehmen, daß der 8086 auf ein Byte mit der Adresse 000F1, einer ungeraden Adresse, zugreift. Der Wert wird dann vom oberen Byte des 16-Bit-Datenwortes an der Adresse 000F0 gelesen. In diesem Fall ist A0 logisch 1 und BHE ist logisch 0.

16-Bit-Zugriff an ungerader Adresse

Bis jetzt haben wir zwei Fälle beschrieben, in denen der 8086 auf Daten im Speicher zugreift. Dabei waren die logischen Zustände von A0 und BHE unterschiedlich. Nun muß folgende Frage gestellt werden: „Was geschieht, wenn der 8086 auf ein 16-Bit-Wort zugreifen muß, das an einer ungeraden Adresse beginnt?" Dieses Problem ist in Abbildung 3.4 dargestellt.

Das ist ein typisches Problem, das auf die Flexibilität der Speicherstruktur des 8086 zurückzuführen ist. Befehle können nämlich eine gerade oder ungerade Anzahl von Bytes benötigen. Damit kein Speicherplatz ver-

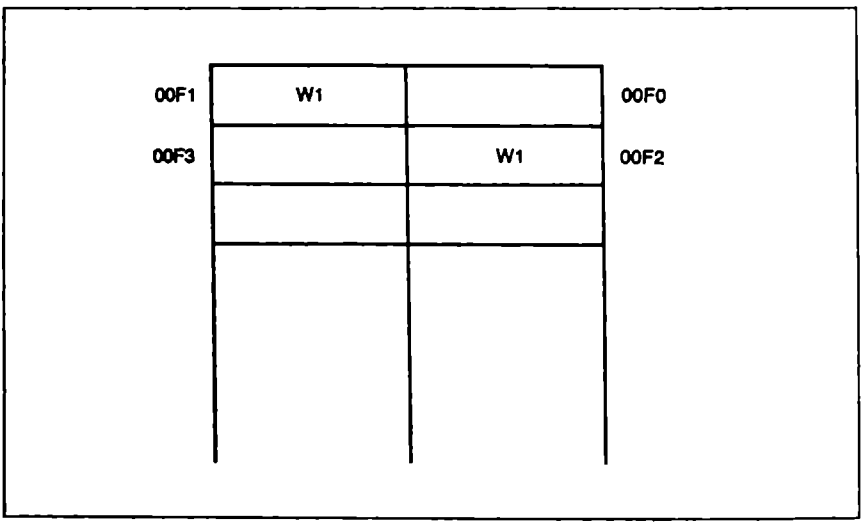


Abb. 3.4: Falls ein Wort an einer ungeraden Adresse gespeichert ist, greift der 8086 auf das gesamte Wort in zwei Speicherzyklen zu

schwendet wird, erlaubt der 8086, daß Datenbytes oder Datenworte sowohl an einer geraden als auch an einer ungeraden Adresse beginnen dürfen. Um ein 16-Bit-Wort zu lesen, das an einer ungeraden Adresse beginnt, muß der 8086 aus diesem Grund zwei Speicherzugriffe durchführen.

AUSGERICHTETE UND NICHTAUSGERICHTETE WÖRTER

Nun kennen wir die verschiedenen Arten, mit denen der 8086 auf Daten im Speicher zugreift. Wenn die CPU auf ein Wort zugreift (d.h. 16 Bits), das an einer geraden Adresse gespeichert ist, greift sie auf ein sogenanntes ausgerichtetes Wort zu. Das Wort ist ausgerichtet, weil beide Bytes in derselben Wortadresse gespeichert sind und in einem einzigen Speicherzyklus gelesen oder geschrieben werden können.

Wenn die CPU auf ein Wort zugreift, das an einer ungeraden Adresse beginnt, spricht man von einem Zugriff auf ein nichtausgerichtetes Wort. Dies ist auf die Tatsache zurückzuführen, daß die beiden Bytes des Wortes nicht in derselben Wortadresse untergebracht sind. Daher sind zwei Speicherzyklen erforderlich, um das ganze Wort zu lesen. Abb. 3.5 zeigt das Konzept ausgerichteter und nichtausgerichteter Wörter.

Der 8086 ist so konstruiert, daß er ausgerichtete und nichtausgerichtete Wörter handhaben kann. Die Ausrichtung bzw. Nichtausrichtung hat

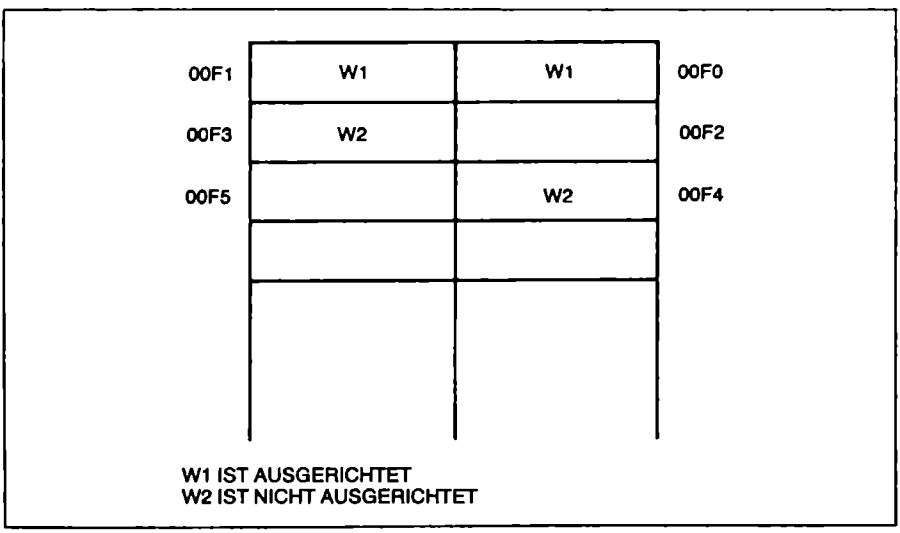


Abb. 3.5: Dieses Diagramm zeigt das Konzept ausgerichteter und nichtausgerichteter Wörter. Auf ausgerichtete Wörter kann in einem einzigen Speicherzyklus zugegriffen werden

Einfluß auf die Ausführungsgeschwindigkeit gewisser Operationen. Wenn Ihr Programm zum Beispiel viele Wörter adressiert oder verarbeitet, so können Sie die Ausführung beschleunigen, indem Sie die Speicherung der Daten an einer geraden Adresse beginnen.

SYSTEMSPEICHERORGANISATION DES 8088

Ebenso wie die 8086-CPU verfügt auch die 8088-CPU über 20 Adreßbits. Ungleich dem 8086 hat sie jedoch nur einen 8-Bit-Datenbus. Dies führt zu einer Speicherorganisation, wie sie in Abb. 3.6 dargestellt ist. Der 8088 ist

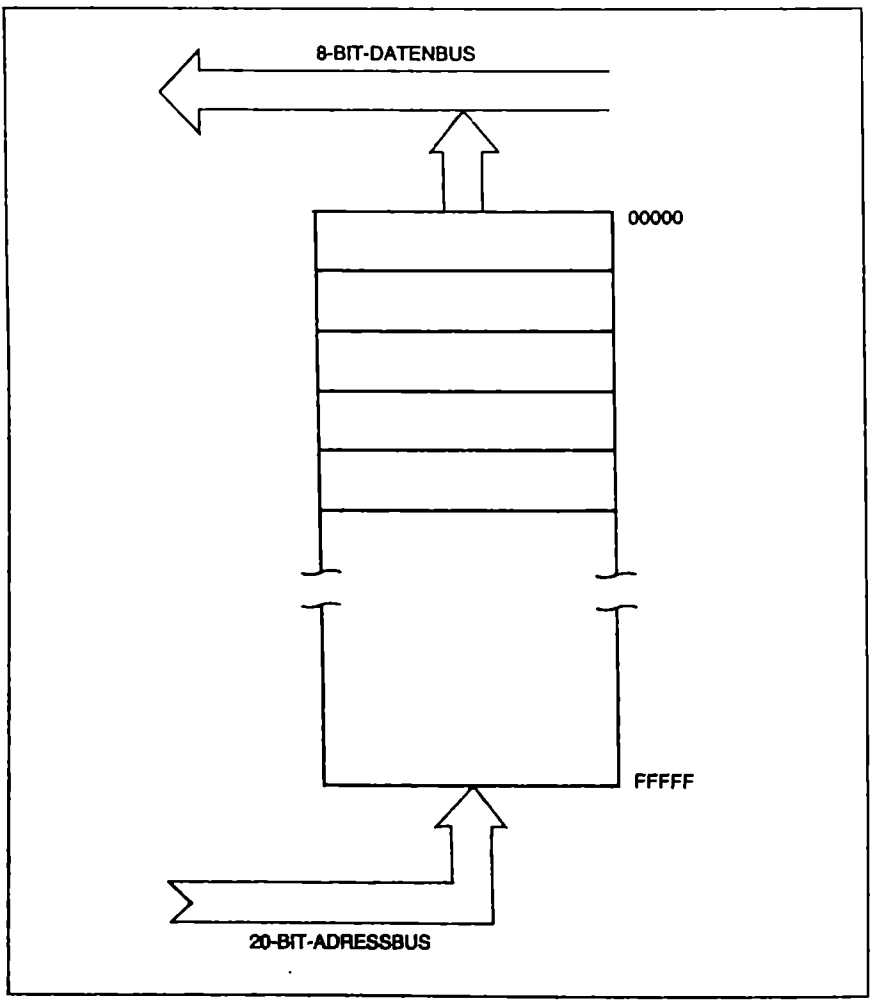


Abb. 3.6: Dieses Blockdiagramm zeigt den 8088-Systemspeicher

im wesentlichen ein 8-Bit-System, das einen 16-Bit-Mikroprozessor verwendet. Sämtliche der 16-Bit-Softwaremöglichkeiten sind auf dem 8088 verfügbar. Dies wird durch die Tatsache demonstriert, daß der 8086 und der 8088 dieselben Softwarebefehle ausführen.

ADRESSERZEUGUNG MIT DEM 8086/8088

In den vorangehenden Erörterungen haben wir die Speicherorganisation für die 8086-CPU und die 8088-CPU vorgestellt. In Kapitel 2 haben wir die internen Register dieser Mikroprozessoren behandelt. In den verbleibenden Abschnitten dieses Kapitels lernen wir, wie Adressen unter Benutzung der internen CPU-Register erzeugt werden. Die unterschiedlichen Arten der Erzeugung einer Adresse werden als *Adressierungsarten* bezeichnet.

Erzeugung von Befehlsadressen

Wir wollen zunächst sehen, wie der 8086/8088 eine Adresse für einen Befehl im Speicher erzeugt. Rufen Sie sich aus den vorangegangenen Betrachtungen ins Gedächtnis zurück, daß alle internen Register auf dem 8086/8088 16 bit breit sind. Der physikalische Adreßbus der CPU ist jedoch 20 bit breit. Das bedeutet, daß mehr als ein internes Register der CPU zur Erzeugung einer physikalischen 20-Bit-Adresse benutzt werden muß.

Die beiden Register, die für eine Befehlsadresse verwendet werden, sind das IP (Befehlsregister) und das CS (Codesegmentregister). Diese beiden Register werden auf eine besondere Weise miteinander verknüpft, um die vollständige 20-Bit-Adresse zu erzeugen.

Hier ist die Gleichung, die beschreibt, wie diese beiden 16- Bit-Register verknüpft werden:

$$\text{20-Bit-Adresse} = (16 \times \text{CS}) + \text{IP}$$

Zum Beispiel:

CS-Register = 1000H
IP-Register = 0414H

Dann folgt:

20-Bit-Adresse = (16 x 1000H) + 0414H
= 10000H (4 Bits nach links verschoben)
+ 0414H

Das ist gleich

10414H.

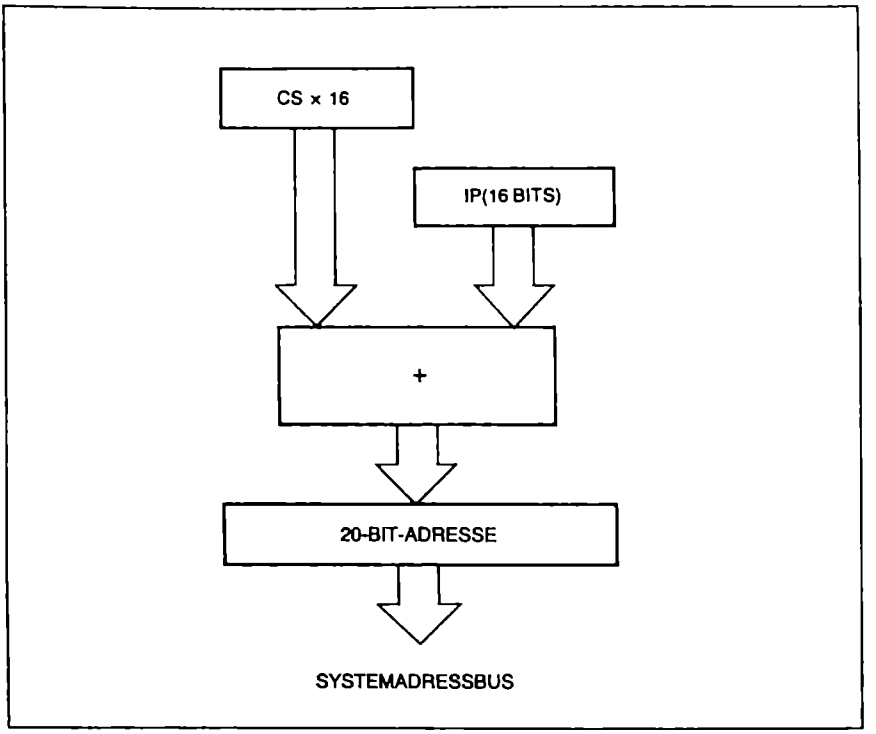


Abb. 3.7: Eine 20-Bit-Systembefehlsadresse wird durch den Ausdruck (CS x 16) + IP berechnet

Dies ist die Adresse im Speicher, von welcher der neue Befehl geholt werden wird. Das IP-Register wird als *Offset* bezeichnet, das CS-Register x 16 zeigt auf die Startadresse oder das Segment im Speicher, worauf das Offset addiert wird. Das bedeutet, es zeigt auf die Startadresse des Speichersegments, das die Befehlscodes enthält; daher der Name *Codesegmentregister* (code segment register). Abbildung 3.7 zeigt in einem Blockdiagramm, wie eine 20-Bit- Befehlsadresse berechnet wird.

Jede Adresse, die vom 8086/8088 erzeugt wird, benutzt eines der vier Segmentregister. Dieses Segmentregister wird dann um vier Bits nach links verschoben, bevor es zu dem Offsetwert addiert wird. Der Offsetwert wird durch Addition der internen Register berechnet. Der CPU-Befehl spezifiziert, welche internen Register zur Erzeugung des Offsets benutzt werden. Beachten Sie, daß nie ein Segmentregister zur Berechnung eines Offsetwertes bei der Adreßerzeugung herangezogen wird.

Nach dieser kurzen Einleitung über die Bildung einer 20- Bit-Adresse lassen Sie uns einige Beispiele verschiedener Typen gebräuchlicher Adres-

sierungsarten untersuchen. Wir werden den 8086/8088-Befehl MOV verwenden, weil er leicht zu verstehen ist. Unser Ziel ist es, mehr über die verschiedenen Adressierungsarten des 8086/8088 zu lernen.

Der MOV-Befehl

Der MOV-Befehl überträgt ein Byte oder ein Wort von einem Quelloperanden zu einem Zieloperanden. (Ein Operand ist ein Wert, der von der CPU verarbeitet wird.) Der Befehl wird geschrieben als:

```
MOV  Ziel, Quelle
```

Unmittelbare Adressierung

Bei der unmittelbaren Adressierung erscheint der Operand im Befehl selbst. Ein Befehl, der einen konstanten Wert in ein internes Register (welches in diesem Fall AX ist) schreibt, ist ein Beispiel für einen Befehl mit unmittelbarer Adressierung:

```
MOV  AX, 568
```

Die Konstante 568 wird in das interne Register AX geladen. Die Zahl 568 ist Teil der Speicherbytes, die den Befehl darstellen.

Registeradressierung

Die Registeradressierungsart gibt an, daß der zu benutzende Operand in einem der allgemeinen internen Register der CPU enthalten ist. Im Fall der AX-, BX-, CX- oder DX-Register kann dieses Register 8 oder 16 bit breit sein. (Diese Register wurden in Kapitel 2 behandelt.) Zum Beispiel bringt der Befehl MOV AX, BX den Inhalt des Registers BX in das Register AX. Ein ähnlicher Befehl, MOV AL, BL bringt den Inhalt von Register BL (8 Bits) in das Register AL. Bei der Verwendung von Registeradressierung führt die CPU alle Operationen intern aus; infolgedessen wird keine 20-Bit-Adresse zur Bestimmung des Operanden erzeugt.

Direkte Adressierung

Bei der direkten Adressierungsart wird im Befehl die Stelle im Speicher, die den Operanden enthält, vollständig bestimmt. Bei dieser Art der Adressierung muß eine 20-Bit-Adresse gebildet werden. Das bedeutet, daß eine interne Adreßerzeugung auf dem 8086/8088 stattfindet. Hier ein Beispiel für diese Adressierungsart:

```
MOV  CX, COUNT
```

Der Wert von COUNT ist eine Konstante. Er wird als Offsetwert bei der Berechnung der 20-Bit-Adresse benutzt. Der 8086/8088 verwendet

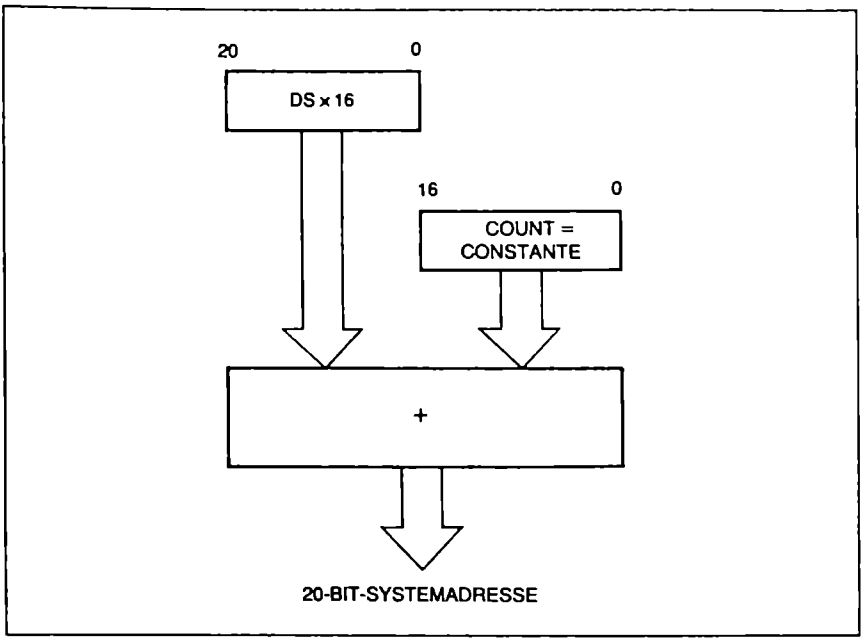


Abb. 3.8: Beim Zugriff auf Daten wird das DS-Register mit einem Offset verknüpft, um die 20-Bit-Systemadresse zu berechnen. Dies ist das voreingestellte Segment. Es kann durch ein Segment-Überschreibungs-Präfix in einem Befehl überschrieben werden

immer ein Segmentregister zur Berechnung einer physikalischen Adresse. Welches Register wird nun für diesen speziellen Befehl verwendet? Wenn es nicht anders angegeben ist, wird das DS- oder Datensegmentregister benutzt (siehe Abb. 3.8). Dies ist das voreingestellte Segment. Es kann jedoch auch jeder der vier Segmentwerte benutzt werden. Dies wird erreicht, indem man das entsprechende Register in dem Befehl angibt. Nehmen wir zum Beispiel an, wir wollen anstelle des DS-Registers lieber das ES-Register benutzen, dann müßten wir schreiben:

```
MOV  CX, ES:COUNT
```

wobei ES das „Segment-Überschreibungs-Präfix" des Befehls ist. Sein Wert wird als Segmentregister zur Berechnung der physikalischen 20-Bit-Adresse benutzt.

Registerindirekte Adressierung

Bei der registerindirekten Adressierungsart ist die 16-Bit-Offsetadresse in einem Basis- oder Indexregister enthalten. Das heißt, die Adresse steht

im BX-, BP-, SI- oder DI-Register. Hier ein Beispiel für diese Adressierungsart.

```
MOV  AX, [SI]
```

Der 16-Bit-Wert, der im SI-Register enthalten ist, wird als Offset zur Berechnung der 20-Bit-Adresse benutzt. Wieder wird ein Segmentregister zur Erzeugung der endgültigen Adresse verwendet. Der 16-Bit-Wert in SI wird mit dem passenden Segmentwert zur Adreßerzeugung verknüpft.

Registerindirekte Adressierung mit Verschiebeanteil

Dieser Typ der Adressierung bezieht die zwei vorangegangenen Adressierungsarten mit ein. Die 16-Bit-Offsetadresse wird durch Addition des 16-Bit-Wertes, der durch das interne Register bestimmt wird, und einer Konstanten berechnet. Wir wollen im folgenden Beispiel das interne Register DI und als Verschiebeanteil einen konstanten Wert, der als COUNT definiert wurde, benutzen. Die mnemonische Darstellung dieses Typs der Adressierung ist

```
MOV  AX, COUNT  [DI]
```

Falls COUNT = 0378H und DI= 04FAH ist, so ist die 16-Bit-Offsetadresse 0872H – die Summe aus 0378H + 04FAH.

Registerindirekte Adressierung mit Basis und Indexregister

Diese Adressierungsart benutzt die Summe zweier interner Register des 8086/8088, um die 16-Bit-Offsetadresse, die zur Berechnung der 20-Bit-Adresse benutzt wird, zu erhalten. Hier zwei Beispiele für diese Adressierungsart:

```
A  MOV  [BP]  [DI], AX
B  MOV AX,  [BX]  [SI]
```

In Beispiel A berechnet sich das Offset der Zieladresse als Summe der Inhalte der Register BP und DI. In Beispiel B ist das Offset der Quelladresse die Summe der Inhalte der Register BX und SI.

Register indirekt mit Basis + Index + Konstante

Dies ist die komplizierteste Adressierungsart. Doch wenn wir sie schrittweise durchgehen, ist sie nicht allzu schwer zu verstehen. Diese Adressierungsart ist identisch mit der vorherigen, mit der Ausnahme, daß eine Konstante zu der sich am Ende ergebenden 16-Bit-Summe hinzuaddiert

wird. Wir wollen zum Beispiel annehmen, daß die folgenden Registerwerte gegeben sind:

DI = 0367H
BX = 7890H
COUNT = 0012H

Der Verschiebeanteil, der sich aus der Summe DI + BX + COUNT ergibt, wird benutzt um ein Wort vom Speicher in das Register AX zu laden. Der Befehl wird folgendermaßen geschrieben:

MOV AX, COUNT [BX] [DI]

Die 16-Bit-Verschiebeadresse ist gleich 7C09H, der Summe aus 0367H + 7890H + 0012H. Die vollständige 20-Bit-Adresse wird berechnet durch die Gleichung (16 x DS) + 7C09H. Wenn z. B. das DS-Register den Wert 3000H hat, ergibt sich die vollständige physikalische 20-Bit-Adresse zu (16 x 3000H) + 7C09H = 37C09H.

8086/8088 OBJEKTCODE

Die oben beschriebenen Adressierungsarten werden vom 8086/8088 benutzt. Als Programmierer werden Sie die mnemonische Form schreiben. Der Objektcode wird sehr häufig vom Computer erzeugt. (Sie erinnern sich, daß der *Objektcode* aus den Datenbytes besteht, die die CPU verarbeitet.) Im verbleibenden Teil dieses Kapitels werden wir im Detail erörtern, wie der Objektcode für die 8086/8088-CPU erzeugt wird. Diese Information wird für Sie beim Studium des 8086/8088Befehlssatzes hilfreich sein. (Eine Zusammenfassung des Befehlssatzes finden Sie in Kapitel 4.)

Wir werden nun die wichtigen Punkte bei der Erzeugung des Objektcodes für den 8086/8088 betrachten. Lassen Sie uns einige Beispiele untersuchen. Diese Beispiele sind die gleichen, die wir schon früher in den Abschnitten über Adressierungsarten vorgestellt haben. In jedem Fall werden wir den Objektcode angeben und eine Erklärung, weshalb die Bits auf logisch 1 oder logisch 0 gesetzt werden.

Bevor wir irgendein Beispiel für den Objektcode untersuchen, sollten wir erklären, daß es schwierig ist, genau zu sagen, wie viele Bytes ein bestimmter Befehl erfordert. (Diejenigen unter Ihnen, die früher 8-Bit-Mikroprozessoren programmiert haben, werden sich erinnern, daß dies nicht immer der Fall gewesen ist.) Beim 8086/8088-Befehlssatz kann, bei ein und demselben Befehl, jede Adressierungsart eine unterschiedliche Anzahl von Datenbytes erfordern. Deshalb werden wir in den folgenden Beispielen nur die Anzahl der Bytes, die für jede Adressierungsart benötigt wird, für jeweils einen ausgewählten Befehl angeben. Ihre Aufgabe wird es sein, zu entscheiden, welche Daten in jedem Byte stehen müssen.

REG-Feld und W-Bit

Das erste Beispiel, das wir betrachten, ist der Befehl MOV AX, 568. Dieser Befehl bewirkt das Laden eines Direktoperanden in ein internes Register. Das interne Register ist das Ziel. Dieses kann ein Byte- oder ein Wortregister sein. Dieser Befehl erfordert 2 oder 3 Bytes, wie in Abbildung 3.9 gezeigt wird.

Wir können in dieser Abbildung erkennen, daß die oberen 4 Bits des ersten Datenbytes bei diesem Befehl 1011 sind. Das nächste Bit ist mit W bezeichnet. Das W steht für Wort =1 oder Byte = 0. Das bedeutet: Wenn das Zielregister ein 16-Bit- oder Wortregister ist, dann ist dieses Bit logisch 1. Falls das Zielregister ein 8-Bit- oder Byteregister ist, so ist dieses Bit logisch 0.

Die nächsten drei Bits des ersten Bytes in der Abb. 3.9 sind mit REG gekennzeichnet. Dieses 3-Bit-Feld bestimmt, in welches Register, Byte- oder Wortregister, die Daten geladen werden. Die 3-Bit-Definitionen für die Register werden in der Abb. 3.10 angegeben.

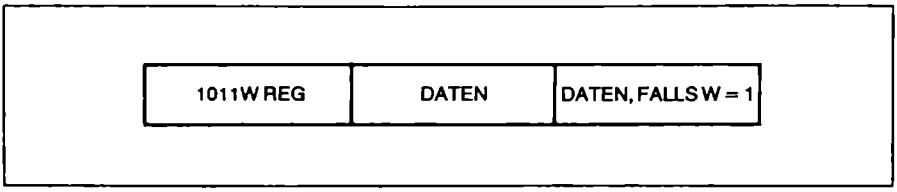


Abb. 3.9: Diese Abbildung zeigt den Objektcode für den MOV Reg, Konstante-Befehl

	16-BIT REGISTER	8-BIT REGISTER
000	AX	AL
001	CX	CL
010	DX	DL
011	BX	BL
100	SP	AH
101	BP	CH
110	SI	DH
111	DI	BH

Abb. 3.10: Diese Abbildung zeigt die Registerdefinition (8- und 16-Bit) für das REG-Feld des Objektcodes

Wenn wir die nächsten zwei Bytes des Befehls in der Abbildung 3.9 untersuchen, sehen wir, daß diese mit DATA gekennzeichnet sind. Falls das Zielregister ein Byte ist, stehen die Daten im zweiten Byte des Befehls. Wenn das Ziel ein Wort ist, ist das zweite Byte des Befehls das untere Byte des 16-Bit-Direktoperanden. Das dritte Byte ist dann das obere Byte des 16-Bit-Direktoperanden.

Für dieses Beispiel (MOV AX, 568) wollen wir annehmen, daß 568 ein hexadezimaler Direktoperand ist. Wir wollen weiter annehmen, daß der Befehl ein 3-Byte-Befehl ist und daß die Bytes B8 68 05 sind. Beachten Sie, daß im ersten Byte das W-Bit gleich 1 und REG gleich 000 ist für AX.

Lautete der Befehl jedoch MOV AL, 56, so wäre es ein 2- Byte-Befehl. Diese Bytes wären B0 56. Das erste Byte hätte das W-Bit gleich 0 und REG gleich 000 für AL gemäß Abbildung 3.10.

D-Bit, MOD und R/M

In diesem Beispiel werden wir Daten aus dem Speicher laden oder ein Register von einem oder zu einem anderen Register laden. Wir werden einen Befehl wie MOV AX, BX benutzen. Dieser Befehl ist ein 2-Byte-Befehl, da keine Speicheradresse anzuhängen ist. Die Bytes haben das Format wie in Abbildung 3.11.

Wir sehen in Abbildung 3.11, daß das erste Byte die unteren zwei Bits DW hat. Das W steht für Wort oder Byte (wie oben). Das D gibt an, ob die Daten in dem Operanden gespeichert werden sollen, der durch das MOD- und R/M-Feld bestimmt wird (D=0), oder ob sie in dem Register gespeichert werden sollen, das durch das REG-Feld bestimmt wird (D=1).

Abb. 3.12 zeigt die MOD- und R/M-Zuordnungen. Beachten Sie, daß das R/M-Feld mit einem Registerformat belegt ist, falls das MOD-Feld den Wert 11 hat. Dieses Format wurde in Abb. 3.10 angegeben.

Bei unserem Beispiel wollen wir die Daten im AX-Register speichern. Das D-Bit wird deshalb logisch 0 sein. Das bedeutet, daß die Daten in der Zielposition gespeichert werden, die durch das MOD- und R/M-Feld spezifiziert wird. Das MOD-Feld muß aus diesem Grunde gleich 11 sein. Das R/M-Feld wird gleich 000 sein, da das AX-Register das Ziel der Daten ist.

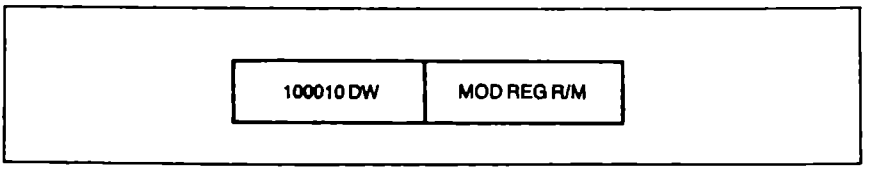


Abb. 3.11: Ein 2-Byte-Register-zu-Register-Befehl

Das REG-Feld des zweiten Datenbytes wird 011, um anzuzeigen, daß das BX Register die zu benutzende Quelle ist (dieser Wert stammt aus Abb. 3.10). Das vollständige zweite Datenbyte des Befehls wird 11011 000 oder D8 sein. Der Objektcode für den Befehl MOV AX,BX ergibt sich so zu 89D8.

BASIS- UND INDEXREGISTER, ANGEGEBEN IN R/M
FÜR OPERANDEN IM SPEICHER (MOD ≠ 11)

R/M FELD	BASISREGISTER	INDEXREGISTER
000	BX	SI
001	BX	DI
010	BP	SI
011	BP	DI
100	NONE	SI
101	NONE	DI
110	BP	NONE
111	BX	NONE

MOD	VERSCHIEBEANTEIL	KOMMENTAR
00	NULL	
01	8-BIT-INHALT DES NÄCHSTEN BYTES DES BEFEHLS MIT AUF 16 BIT ERWEITERTEM VORZEICHEN	DER BEFEHL ENTHÄLT EIN WEITERES BYTE
10	16-BIT-INHALT DER NÄCHSTEN ZWEI BYTES DES BEFEHLS	DER BEFEHL ENTHÄLT WEITERE ZWEI BYTES
11	R/M-Register	

FALLS MOD = 00 UND R/M = 110, DANN:
1. OBIGE AUFSTELLUNG GILT NICHT
2. DER BEFEHL ENTHÄLT ZWEI WEITERE BYTES
3. DER VERSCHIEBEANTEIL DER ADRESSE IST IN DIESEN BYTES ENTHALTEN

Abb. 3.12: R/M-Feld und MOD-Definitionen für den 8086/8088-Objekt-Code

OBJEKTCODE FÜR INDIZIERUNG UND BENUTZUNG VON BASISREGISTERN

Wir wollen ein letztes Beispiel der Erzeugung von Objektcode für den 8086/8088 betrachten. In diesem Beispiel werden wir den Objektcode für den Befehl MOV BX, COUNT [BX] [SI] berechnen. Dieser Befehl hat das gleiche allgemeine Format wie das vorangegangene Beispiel. Die Anzahl der Bytes für diesen Befehl ist 4. Diese Bytes werden in Abbildung 3.13 dargestellt.

Beim ersten Byte in Abb. 3.13 ist das D-Bit auf logisch 1 gesetzt. Das entspricht der Tatsache, daß das Ziel der Daten durch das REG-Feld im zweiten Byte bestimmt wird. Das W-Bit ist auf logisch 1 gesetzt, weil es sich um eine Wortübertragung handelt. Das gibt als erstes Datenbyte 10001011 oder 8B.

Als nächstes gehen wir zum MOD-Feld des zweiten Bytes. Da wir eine Konstante benutzen, die 16 Bits in Anspruch nimmt, ist MOD auf 10 gesetzt. Wie aus Abb. 3.12 ersichtlich, zeigt dies an, daß der Verschiebeanteil in zwei Bytes formatiert ist und diesem Byte folgt.

Das nächste Feld des zweiten Bytes ist das Registerfeld oder REG. Da wir das CX-Register benutzen, ist der Wert auf 001 gesetzt (dies erhielten wir von der Registerfelddefinition, die in Abbildung 3.10 angegeben ist).

Schließlich haben wir das R/M-Feld. Da das MOD-Feld nicht gleich 11 war, spezifiziert dieses Feld, welche Basis- und Indexregister benutzt werden, um die 16-Bit-Offsetadresse zu erzeugen. In unserem Fall benutzen wir das (BX + SI + DISP)- Feld. Das entspricht einem R/M von 000 (siehe Abb. 3.12).

Das gesamte zweite Byte der Daten ist gleich 10001000. Das ist gleich 8 H. Das dritte und vierte Byte ist gleich dem Verschiebeanteil. In diesem Fall ist der Wert von COUNT 0345 hexadezimal. Die letzten beiden Bytes sind 4503. Dies ergibt als vollständigen Objektcode für den Befehl MOV CX, COUNT [BX] [SI] die Folge von Hexadezimalziffern 8B 88 45 03.

ZUSAMMENFASSUNG OBJEKTCODE

Eine Frage, die Sie möglicherweise an diesem Punkt stellen ist „Werde ich die D-, W-, REG-, MOD-, und R/M-Felder stets auszufüllen haben?". Falls dies der Fall wäre, würden Sie wahrscheinlich den 8086/8088 nicht programmieren wollen.

Glücklicherweise wird Ihr Computer diese Arbeit für Sie erledigen. Dieser Objektcodeabschnitt wurde nur aufgenommen, um Sie zu einem besseren Verständnis der inneren Arbeitsweise des 8086/8088-Mikroprozessors zu befähigen.

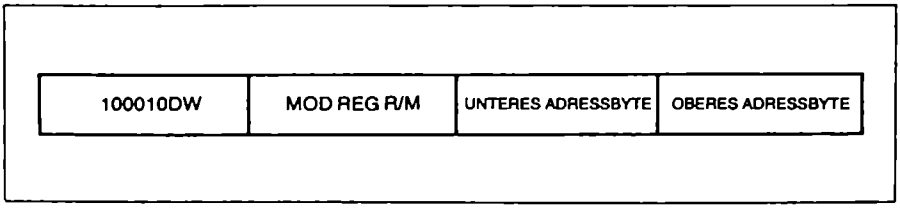


Abb. 3.13: Das Objektcodeformat für einen Befehl wie MOV BX, COUNT [BX] [SI]

ZUSAMMENFASSUNG

In diesem Kapitel haben wir einige wichtige Punkte im Zusammenhang mit der Programmierung des 8086/8088 betrachtet. Wir begannen mit einer Darstellung der Systemspeicherorganisation für beide CPU. Diese Betrachtung wurde angestellt, um Ihnen zu helfen, klarer zu verstehen, wie die Befehle im Systemspeicher organisiert sind; weiter erlaubt sie Ihnen, die Unterschiede und Ähnlichkeiten zwischen den 8086/8088-Systemen zu sehen.

Als nächstes diskutierten wir sämtliche Adressierungsarten. Bei der Darstellung jeder Art wurden Beispiele gegeben, um exakt darzustellen, wie die Adressen erzeugt werden. Wir haben auch die Segment-, Basis- und Indexregister betrachtet und gelernt, wie eine physikalische 20-Bit-Adresse intern von der CPU erzeugt wird.

Wir schlossen das Kapitel mit Beispielen, die zeigten, wie man Objektcode für den 8086/8088 erzeugt. Diese Beispiele schlossen Betrachtungen der verschiedenen Felder, die den Objektcode ausmachen, mit ein, einschließlich der D-, W-, R/M-, REG- und MOD-Felder.

Die Information, die in diesem Kapitel gegeben wurde, soll Ihnen helfen, die Codierung der Befehle, die im nächsten Kapitel gezeigt wird, besser zu verstehen.

Kapitel 4

Der Befehlssatz: Individuelle Beschreibungen

In diesem Kapitel präsentieren wir den vollständigen 8086/8088-Befehlssatz alphabetisch nach den mnemonischen Darstellungen aufgeführt. Wir geben eine kurze Beschreibung eines jeden Befehls und zeigen seinen Gebrauch in der Assemblersprache (einschließlich der Operanden). Zusätzlich beschreiben wir, wie der 8086/8088 jeden Befehl ausführt. Dies ist hilfreich, wenn Sie eine schnelle Referenz haben wollen, die aufzeigt, was während der Ausführung eines Befehls tatsächlich passiert. Jede Beschreibung schließt die Besonderheiten, die ein einzelner Befehl haben mag, mit ein, und es wird klar dargestellt, wann ein Befehl verwendet werden kann und wann nicht.

Für bestimmte Befehle werden Beispiele angegeben. Für andere ist von der Beschreibung her klar, wie sie benutzt werden müssen.

Schließlich wird für jeden Befehl der Objektcode angegeben.

Die mnemonische Darstellung, die in diesem Kapitel verwendet wird, ist identisch mit der von der Firma Intel entwickelten. Die hier gegebene Information soll Ihnen nicht nur helfen, besser zu verstehen, wie der 8086/8088 programmiert werden kann, sondern soll Ihnen auch als wertvolle Referenz dienen, wenn Sie beginnen, Programme in Assemblersprache zu schreiben.

Es gibt von verschiedenen Herstellern unterschiedliche Assembler für den 8086/8088. Die Unterschiede liegen nicht in den von der Firma Intel festgelegten mnemonischen Bezeichnungen für die einzelnen Befehle, sondern in den zur Verfügung stehenden Pseudobefehlen (Direktiven) zur Definition von Symbolen, Datenbereichen, Segmenten usw... Es würde den Rahmen dieses Buches sprengen, die Gesamtheit dieser Direktiven darzustellen oder die unterschiedlichen Implementierungen aufzuzählen. Wichtig für das Verständnis der Beispiele in diesem Kapitel ist es jedoch zu wissen, daß jedem vom Benutzer definierten Symbol *Attribute* zugeordnet werden, die der Assembler benutzt, um die richtige Adressierungsform in einem Befehl zu wählen bzw. um im Einzelfall zu prüfen, ob der jeweilige Befehl mit dem Operanden erlaubt ist oder der

Operand über die aktuellen Inhalte der Segmentregister überhaupt adressiert werden kann. Auf diese Weise kann der Assembler dem Benutzer die Schreibarbeit erleichtern und auch sonst schwer zu findende Programmfehler aufdecken. So unterscheidet der Assembler bei benutzerdefinierten Symbolen Vereinbarungen von

- Konstanten,
- Marken für Sprungziele innerhalb und außerhalb des aktuellen Codesegments,
- Marken für Wertvereinbarungen oder Speicherreservierungen innerhalb und außerhalb des aktuellen Datensegments (bei Zeichenkettenbefehlen auch des aktuellen Extrasegments).

Bei Marken für Wertvereinbarungen wird weiter unterschieden, ob Speicher für Bytes, Wörter oder Doppelwörter vereinbart bzw. reserviert wurde.

Diese Attribute benutzt der Assembler zum Beispiel, um bei einem CALL-Befehl zu entscheiden, ob ein Inter- oder ein Intrasegment-CALL zu erzeugen ist. Aus dem Namen des Sprungziels allein kann dies nicht entnommen werden. Falls Sie nur mit 8-Bit-Mikroprozessoren und deren Assemblern vertraut sind, werden für Sie die weiteren Hinweise in Kapitel 9 hilfreich sein.

Die Tafel unten zeigt die Abkürzungen und Symbole. Abbildung 4.1 enthält ein Modell der 8086/8088-Register, das wir bei der folgenden Beschreibung der 8086/8088-Befehle verwenden.

Abkürzungen und Symbole für Befehlsbeschreibungen

Falls d=1, dann „zu"; falls d=0, dann „von".
Falls w=1, dann Wort-Befehl; Falls w=0, dann ByteBefehl.
Falls s:w=01, dann bilden 16 Bits unmittelbar folgender Daten den Operanden.
Falls s:w=11, dann wird ein unmittelbar folgendes Datenbyte vorzeichenerweitert, um den 16-Bit-Operanden zu bilden.
Falls v=0, dann „count"=1; sonst „count" in CL.
x = keine Bedeutung.
z wird für einige Zeichenkettenbefehle benutzt, um Vergleiche mit dem ZF-Kennzeichen durchzuführen.
AL = 8-Bit-Akkumulator.
AX = 16-Bit-Akkumulator.
CX = Zähl-Register.
DX = Register für variable Anschlußadresse.
DS = Datensegment.
ES = Extrasegment.

Abkürzungen und Symbole für Befehlsbeschreibungen (Forts.)

Größer = größere positive Zahlen mit Vorzeichen.
Kleiner = kleinere positive (betragsgrößere negative) Zahlen mit Vorzeichen.

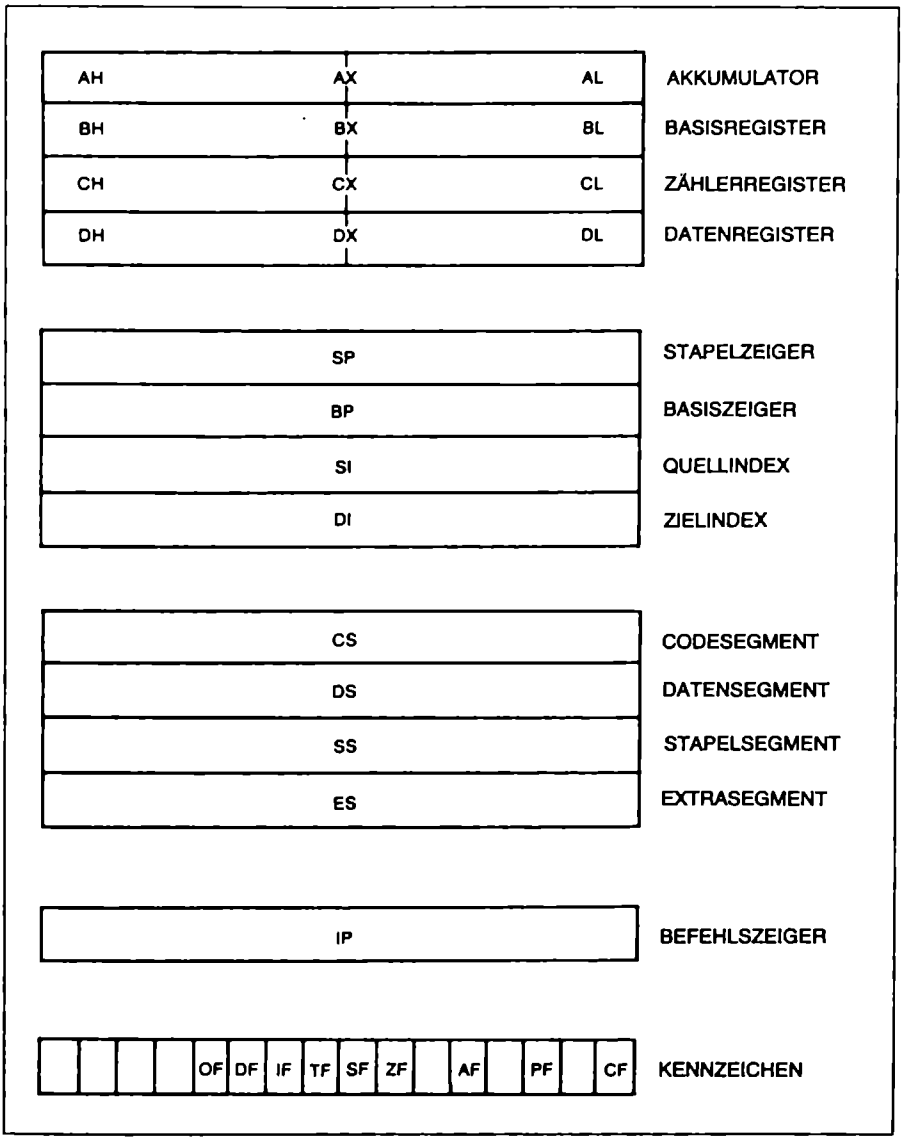


Abb. 4.1: Die internen Register des 8086/8088

AAA *ASCII Adjust For Addition*
ASCII-Anpassung für Addition

Mnemonische Darstellung: AAA keine Operanden

Funktion: Falls D3,D2,D1,D0 von AL > 9 oder AF Kennzeichenbit=1, dann AL = AL + 6, AH = AH + 1, AF = 1, CF = AF, AL = AL ∧ 0FH

Definierte Kennzeichenbits: AF, CF

Undefinierte Kennzeichenbits: OF, PF, SF, ZF

Beschreibung: Der AAA-Befehl verändert den Wert des AL-Registers zu einer gültigen ungepackten Dezimalzahl; die Bits D7-D4 des AL-Registers werden gelöscht.

Beispiel: AAA

Vorher:	Nachher:
AL = 0BH	AL = 01H
AH = 00H	AH = 01H

Codierung:

00110111

AAD

ASCII Adjust For Division
ASCII-Anpassung für Division

Mnemonische Darstellung: AAD keine Operanden

Funktion: AL = (AH * 0AH) + AL; AH = 0

Definierte Kennzeichenbits: PF, SF, ZF

Undefinierte Kennzeichenbits: AF, CF, OF

Beschreibung: Der AAD-Befehl verändert den Inhalt des AL- und AH Registers (von denen jedes eine gültige ungepackte BCD-Zahl enthält) zu einer äquivalenten binären Zahl, die im AL-Register enthalten ist. Dies ermöglicht dem Programmierer, den IDIV-Befehl zu verwenden, um das korrekte Ergebnis zu erzielen. Das AH-Register muß vor dem IDIV-Befehl gleich Null sein. Nach dem IDIV-Befehl ist der Quotient in AL und der Rest in AH enthalten. Die beiden höherwertigen Halbbytes werden auf Null gesetzt.

Beispiel: AAD

Vorher:	Nachher:
AL = 03H	AL = 35H
AH = 05H	AH = 00H

Codierung:

11010101	00001010

AAM *ASCII Adjust For Multiply*
ASCII-Anpassung für Multiplikation

Mnemonische Darstellung: AAM keine Operanden

Funktion: AH = AL/0AH; AL = Rest

Definierte Kennzeichenbits: PF, SF, ZF

Undefinierte Kennzeichenbits: AF, CF, OF

Beschreibung: Der AAM-Befehl korrigiert das Ergebnis einer Multiplikation zweier gültiger ungepackter BCD-Zahlen. Eine gültige ungepackte BCD-Zahl wird aus dem Inhalt von AH und AL gebildet. Die binäre Zahl in AL wird durch 10 dividiert und der Quotient im AH-Register gespeichert. Der bei der Division gebildete Rest wird im AL-Register gespeichert. Die höherwertigen Halbbytes der multiplizierten Operanden mußten 0h sein, damit AAM ein korrektes Ergebnis liefern kann.

Beispiel: AAM

Vorher:	Nachher:
AH = 00H	AH = 06H
AL = 41H	AL = 05H

Codierung:

11010100	00001010

AAS

ASCII Adjust For Subtraction
ASCII-Anpassung für Subtraktion

Mnemonische Darstellung: AAS keine Operanden

Funktion: Falls D3,D2,D1,D0 von AL > 9 oder AF-Kennzeichenbit = 1, dann AL = AL − 6, AH = AH − 1, AF = 1,
CF = AF, AL = AL ∧ 0FH

Definierte Kennzeichenbits: AF, CF

Undefinierte Kennzeichenbits: OF, PF, SF, ZF

Beschreibung: AAS korrigiert das Ergebnis einer vorangegangenen Subtraktion zweier ungepackter dezimaler Operanden. AAS ändert den Inhalt von AL zu einer gültigen ungepackten Dezimalzahl. Das höherwertige Halbbyte wird auf null gesetzt.

Codierung:

00111111

ADC *Add With Carry*
Mit Übertrag addieren

Mnemonische Darstellung: ADC Ziel, Quelle

Funktion: Falls CF = 1, dann Ziel = Ziel + Quelle + 1
Falls CF = 0, dann Ziel = Ziel + Quelle

Definierte Kennzeichenbits: AF, CF, OF, PF, SF, ZF

Beschreibung: ADC addiert die Operanden, die Bytes oder Wörter sein können und addiert zusätzlich eine Eins, falls CF gesetzt ist. Der Zieloperand wird durch das Ergebnis überschrieben. Beide Operanden können Binärzahlen mit oder ohne Vorzeichen sein.

Beispiele:
```
ADC  AX,BX
ADC  AL, 9
ADC  BX, 458
```

Codierung: Speicher- oder Registeroperand mit Registeroperand

000100dw	mod reg r/m

Unmittelbarer Operand zu Speicher- oder Registeroperand

100000sw	mod 010 r/m	Daten	Daten, falls s:w = 01

Unmittelbarer Operand zu Akkumulator

0001010w	Daten	Daten, falls w = 1

ADD *Addition*
Addieren

Mnemonische Darstellung: ADD Ziel, Quelle

Funktion: Ziel = Ziel + Quelle

Definierte Kennzeichenbits: AF, CF, OF, PF, SF, ZF

Beschreibung: Die Summe der beiden Operanden, die Bytes oder Wörter sein können, ersetzt den Zieloperanden.

Beispiele:
```
ADD  DX,CX
ADD  AX, 400
ADD  Marke,BL
```

Codierung: Speicher- oder Registeroperand mit Registeroperand

000000dw	mod reg r/m

Unmittelbarer Operand zu Speicher- oder Registeroperand

100000sw	mod 000 r/m	Daten	Daten, falls s:w = 01

Unmittelbarer Operand zu Akkumulator

0000010w	Daten	Daten, falls w = 1

AND *And Logical*
Logisches Und bilden

Mnemonische Darstellung: AND Ziel, Quelle

Funktion: Ziel = Ziel ∧ Quelle
CF = 0, OF = 0

Definierte Kennzeichenbits: CF, OF, PF, SF, ZF

Undefinierte Kennzeichenbits: AF

Beschreibung: AND führt das logische „Und" (∧) zwischen den beiden Operanden (Bytes oder Wörter) durch und gibt das Ergebnis zum Zieloperanden zurück. Ein Bit des Ergebnisses wird auf logisch 1 gesetzt, falls beide Bits der ursprünglichen Operanden 1 sind; im anderen Fall wird das Bit gelöscht.

Beispiel: AND BX,CX

Vorher:	Nachher:
BX = AC07H	BX = 2004H
CX = 23F4H	CX = 23F4H

Codierung: Speicher- oder Registeroperand mit Registeroperand

001000dw	mod reg r/m

Unmittelbarer Operand zu Speicher- oder Registeroperand

1000000w	mod 100 r/m	Daten	Daten, falls w = 1

Unmittelbarer Operand zu Akkumulator

0010010w	Daten	Daten, falls w = 1

CALL *Call Procedure* *Prozedur aufrufen*

Mnemonische Darstellung: CALL Prozedurname

Funktion: Falls Intersegment, dann SP = SP − 2; CS auf Stapel retten
CS = spezifiziertes neues Segment; SP = SP − 2;
IP auf Stapel retten;
IP = spezifizierter neuer Befehlszeiger

Falls Intrasegment, dann SP = SP −2; IP auf Stapel retten
IP = spezifierter neuer Befehlszeiger

Beeinflußte Kennzeichenbits: keine

Beschreibung: CALL aktiviert eine „außerhalb der Zeilen" liegende Prozedur und rettet Informationen auf den Stapel, um einem späteren RET-Befehl (Rücksprung) in der Prozedur zu gestatten, die Kontrolle an den Befehl, der dem CALL folgt, zurückzugeben. Der Assembler erzeugt unterschiedliche Typen des CALL-Befehls in Abhängigkeit davon, ob der Programmierer den Prozedurnamen als NEAR (nah) oder FAR (fern) definiert hat. Der Typ des CALL-Befehls muß dem Typ des die Prozedur verlassenden RET-Befehls entsprechen, damit die korrekte Rückgabe der Kontrolle gewährleistet ist.

Bei einem intrasegment-direkten CALL wird SP um 2 erniedrigt und IP auf dem Stapel abgelegt. Der relative Verschiebeanteil (bis zu ± 32K) der Zielprozedur wird dann zum Befehlszeiger addiert.

Ein intrasegment indirekter CALL kann über den Speicher oder ein Register erfolgen. SP wird um 2 erniedrigt und IP auf dem Stapel abgelegt. Das Offset der Zielprozedur, mit dem IP geladen wird, erhält man aus dem im Befehl referierten Speicherwort oder allgemeinen Register.

Bei einem intersegment-direkten CALL wird SP um 2 erniedrigt und CS auf dem Stapel abgelegt. CS wird

durch das in dem Befehl enthaltene Segmentwort ersetzt. SP wird wieder um 2 erniedrigt, und IP wird auf dem Stapel abgelegt und durch das Offsetwort in dem Befehl ersetzt.

Bei einem intersegment-indirekten CALL wird SP um 2 erniedrigt und CS auf dem Stack abgelegt. CS wird dann ersetzt durch den Inhalt des zweiten Wortes des durch den Befehl im Speicher referierten Doppelwortes. SP wird wieder um 2 erniedrigt, und IP wird auf dem Stapel abgelegt und ersetzt durch den Inhalt des ersten Wortes des durch den Befehl im Speicher referierten Doppelwortes.

Beispiele:

CALL NEAR
CALL FAR
CALL AX

Codierung:

Intrasegment-direkt

11101000	disp-low	disp-high

Ziel = Effektive Adresse

Intrasegment-indirekt

11111111	mod 010 r/m

Ziel = IP + Verschiebeanteil

Intersegment-direkt

10011010	offset-low	offset-high	seg-low	seg-high

Ziel = Offset, SEGMENT = Segment

Intersegment-indirekt

11111111	mod 011 r/m

Ziel = Effektive Adresse, Segment = Effektive Adresse +2

CBW

Convert Byte To Word
Byte zu Wort umwandeln

Mnemonische Darstellung: CBW keine Operanden

Funktion: Falls AL < 80H, dann AH = 00H
Falls AL ≥ 80H, dann AH = FFH

Beeinflußte Kennzeichenbits: keine

Beschreibung: CBW pflanzt das Vorzeichen des Bytes in Register AL durch das Register AH hindurch fort.

Beispiel: CBW

Vorher:	Nachher:
AL = 83H	AH = FFH: AL = 83H
AL = 56H	AH = 00H: AL = 56H

Codierung:

10011000

CLC *Clear Carry* *Übertragskennzeichen löschen*

Mnemonische Darstellung: CLC keine Operanden

Funktion: CF = 0

Definierte Kennzeichenbits: CF

Beschreibung: CLC setzt das CF-Kennzeichenbit (carry flag) auf Null.

Codierung:

11111000

CLD

Clear Direction Flag
Richtungskennzeichen löschen

Mnemonische Darstellung: CLD keine Operanden

Funktion: DF = 0

Definierte Kennzeichenbits: DF

Beschreibung: CLD setzt DF auf Null und bewirkt damit, daß Befehle für Zeichenketten das SI- und/oder das DI- Register automatisch erhöhen.

Codierung:

11111100

CLI

Clear Interrupt Enable Flag
Unterbrechungsfreigabekennzeichen löschen

Mnemonische Darstellung: CLI keine Operanden

Funktion: IF = 0

Definierte Kennzeichenbits: IF

Beschreibung: CLI setzt das IF-Kennzeichenbit auf Null, wodurch maskierbare Unterbrechungen gesperrt werden. Software-Unterbrechungen werden nicht gesperrt.

Codierung:

11111010

CMC *Complement Carry Flag*
Übertragskennzeichen komplementieren

Mnemonische Darstellung: CMC keine Operanden

Funktion: Falls CF = 0, dann CF = 1; falls CF = 1, dann CF = 0

Definierte Kennzeichenbits: CF

Beschreibung: CMC bewirkt das „Umklappen" von CF in seinen entgegengesetzten Zustand.

Codierung:

11110101

CMP *Compare* *Vergleichen*

Mnemonische Darstellung: CMP Ziel, Quelle

Funktion: Ziel – Quelle

Definierte Kennzeichenbits: AF, CF, OF, PF, SF, ZF

Beschreibung: CMP subtrahiert die Quelle vom Ziel, aber speichert das Ergebnis nicht ab. Die Operanden bleiben unverändert, aber die Kennzeichenbits werden aktualisiert und können von einem nachfolgenden bedingten Sprungbefehl getestet werden. Quelle und Ziel können Bytes oder Wörter sein.

Beispiele:
```
CMP DX,CX
CMP AL, 25
CMP BH,Marke
```

Codierung: Speicher- oder Registeroperand mit Registeroperand

001110dw	mod reg r/m

Unmittelbarer Operand mit Speicher- oder Registeroperand

100000sw	mod 111 r/m	Daten	Daten, falls s:w = 01

Unmittelbarer Operand mit Akkumulator

0011110w	Daten	Daten, falls w = 1

CMPS *Compare String (Byte or Word)*
Zeichenkette (Byte- oder Wort-) vergleichen

Mnemonische Darstellung: CMPS (Ziel-Zeichenkette) , (Quell-Zeichenkette)

Funktion: (Ziel-Zeichenkette) – (Quell-Zeichenkette)
Falls DF = 0, dann SI = SI + Delta, DI = DI + Delta.
Falls DF = 1, dann SI = SI – Delta, DI = DI – Delta.

Falls die Zeichenkette aus Bytes besteht, ist Delta = 1. Falls die Zeichenkette aus Wörtern besteht, ist Delta = 2.

Die Ziel-Zeichenkette wird durch DI adressiert.
Die Quell-Zeichenkette wird durch SI adressiert.

Definierte Kennzeichenbits: AF, CF, OF, PF, SF, ZF

Beschreibung: CMPS (Compare String) subtrahiert das Zielbyte oder Zielwort (adressiert durch DI) vom Quellbyte oder -wort (adressiert durch SI). CMPS beeinflußt die Kennzeichenbits, aber verändert keinen der Operanden. Falls CMPS ein REPE oder REPZ vorangestellt ist, wird der Befehl interpretiert als „vergleiche, solange nicht Ende der Zeichenkette (CX nicht Null) und die Zeichenketten gleich sind (ZF = 1)." Falls CMPS ein REPNE oder REPNZ vorangeht, wird die Operation interpretiert als "vergleiche, solange nicht Ende der Zeichenkette (CX nicht Null) und die Zeichenketten nicht gleich sind (ZF = 0)." Auf diese Weise kann CMPS verwendet werden, um übereinstimmende und unterschiedliche Zeichenkettenelemente zu finden.

Die in dem CMPS-Befehl angegebenen Operanden werden vom Assembler nur benutzt, um Typ und Zugriffsmöglichkeit zu überprüfen unter Verwendung der laufenden Segmentregisterinhalte.

Codierung:

1010011w

CWD *Convert Word To Double Word*
Wort zu Doppelwort umwandeln

Mnemonische Darstellung: CWD keine Operanden

Funktion: Falls AX < 8000H, dann DX = 0
Falls AX ⩾ 8000H, dann DX = FFFFH

Beeinflußte Kennzeichenbits: Keine

Beschreibung: CWD setzt das Vorzeichen des Wortregisters AX durch das Register DX hindurch fort.

Beispiel: CWD

Vorher:	Nachher:
AX = 9034H	DX = FFFFh, AX = 9034H
AX = 7034H	DX = 0000h, AX = 7034H

Codierung: 10011001

DAA *Decimal Adjust For Addition*
Dezimale Anpassung für Addition

Mnemonische Darstellung: DAA keine Operanden

Funktion: Falls D3,D2,D1,D0 von AL > 9 oder AF-Kennzeichenbit = 1,
dann AL = AL + 6; AF = 1
Falls AL > 9FH oder CF-Kennzeichenbit = 1,
dann AL = AL + 60H; CF = 1

Definierte Kennzeichenbits: AF, CF, PF, SF, ZF

Undefinierte Kennzeichenbits: OF

Beschreibung: DAA ändert den Inhalt von AL zu einem Paar von gültigen gepackten Dezimalzahlen.

Beispiel: DAA

Vorher:	Nachher:
AL = 8AH	AL = 90H

Codierung:

00100111

DAS

Decimal Adjust For Subtraction
Dezimale Anpassung für Subtraktion

Mnemonische Darstellung: DAS keine Operanden

Funktion: Falls D3,D2,D1,D0 von AL > 9 oder AF = 1, dann AL = AL − 6; AF = 1
Falls AL > 9FH oder CF = 1, dann AL = AL − 60H; CF = 1

Definierte Kennzeichenbits: AF, CF, PF, SF, ZF

Undefinierte Kennzeichenbits: OF

Beschreibung: DAS ändert den Inhalt von AL zu einem Paar gültiger gepackter Dezimalzahlen, die das Ergebnis einer vorangegangenen Subtraktion sind.

Beispiel: DAS

Vorher:	Nachher:
AL = 1FH	AL = 19H

Codierung:

00101111

DEC *Decrement*
Dekrementieren

Mnemonische Darstellung: DEC Ziel

Funktion: Ziel = Ziel − 1

Definierte Kennzeichenbits: AF, OF, PF, SF, ZF

Beschreibung: DEC subtrahiert 1 vom Zieloperanden. Der Operand kann ein Byte oder ein Wort sein.

Beispiel: DEC BX

Vorher:	Nachher:
BX = 1200H	BX = 11FFH

Codierung: Speicher- oder Registeroperand

1111111w	mod 001 r/m

Registeroperand

01001 reg

DIV *Divide*
Dividieren

Mnemonische Darstellung: DIV Quelle

Funktion: Falls der Quelloperand ein Byte ist: AX/Quelle:
AH = Quotient, AL = Rest
Falls der Quelloperand ein Wort ist: AX,DX/Quelle,
AX = Quotient, DX = Rest, der die signifikantesten 16 Bits der Division enthält

Definierte Kennzeichenbits: keine

Undefinierte Kennzeichenbits: AF, CF, OF, PF, SF, ZF

Beschreibung: DIV (Divide) führt eine vorzeichenlose Division des Akkumulators und seiner Erweiterung durch den Quelloperanden durch. Falls der Quelloperand ein Byte ist, wird der Dividend mit doppelter Länge in den Registern AL und AH gespeichert. Der Rest mit einfacher Länge wird in AH zurückgegeben. Falls der Quelloperand ein Wort ist, wird der Dividend mit doppelter Länge in den Registern AX und DX gespeichert. Der Rest einfacher Länge wird in DX zurückgegeben. Falls der Quotient die Kapazität seines Zielregisters überschreitet (FFh für Byte-Quelle und FFFFh für Wort-Quelle), sowie, wenn eine Division durch 0 auftritt, wird eine Typ-0-Unterbrechung erzeugt. Zu diesem Zeitpunkt sind Quotient und Rest nicht definiert. Nichtganzzahlige Quotienten werden zu ganzen Zahlen abgeschnitten.

Beispiele:

DIV 25 ;DIES IST EIN BYTE

Vorher:	Nachher:
AH = 00H	AH = 01H
AL = 33H	AL = 02H

DIV 300 ;DIES IST EIN WORT

Vorher:	Nachher:
DX = 0000H	DX = 0005H Rest
AX = 0389H	AX = 0003H Quotient

DIV CL
DIV CX

Codierung:

1111011w	mod 110 r/m	1–2 Bytes können folgen

ESC

Escape
Objektcode und Operand für Koprozessor bereitstellen

Mnemonische Darstellung: ESC externer Opcode, Adresse

Funktion: Falls MOD ungleich 11 ist, wird der Befehl auf den Datenbus gelegt.
Falls MOD gleich 11 ist, wird ein „Befehl ohne Wirkung" (no-operation) ausgeführt.

Beeinflußte Kennzeichenbits: keine

Beschreibung: Der ESC-Befehl stellt einen Mechanismus zur Verfügung, mit dem andere Prozessoren (Koprozessoren) ihre Befehle aus dem 8086/8088-Befehlsstrom empfangen können und gleichzeitig von den Adressierungsarten des 8086/8088 Gebrauch machen können. Die CPU führt keine andere Operation für den ESC-Befehl aus als den Speicherzugriff und stellt den Operanden auf dem Datenbus zur Verfügung. Der 8086/8088 ist entworfen, um mit einer Auswahl von Koprozessoren zu operieren, die unterschiedliche Systemfunktionen ausführen. Zwei Koprozessoren, die mit dem 8086/8088 arbeiten, sind der 8089-Ein-/Ausgabeprozessor und der 8087-Numerikprozessor.

Beispiel: ESC 6, AL

Codierung:

11011xxx	mod xxx r/m

HLT *Halt*
Halten

Mnemonische Darstellung: HLT keine Operanden

Funktion: keine

Beeinflußte Kennzeichenbits: keine

Beschreibung: HLT überführt den 8086/8088 in den Haltezustand. Der Prozessor verläßt den Haltezustand nach der Aktivierung des RESET-Eingangs, infolge des Empfangs einer nichtmaskierbaren Unterbrechungsanforderung an NMI oder, falls Unterbrechungen zugelassen sind, infolge des Empfangs einer maskierbaren Unterbrechungsanforderung an INTR.

Codierung:

11110100

IDIV *Integer Divide* *Ganzzahlig dividieren*

Mnemonische Darstellung: IDIV Quelle

Funktion: Falls der Quelloperand ein Byte ist, wird der Quotient in AL und der Rest in AH gespeichert. Der Dividend ist in AX enthalten und wird durch den angegebenen Quelloperanden dividiert.

Falls der Quelloperand ein Wort ist, wird der Quotient in AX und der Rest in DX gespeichert. Der Dividend ist in AX,DX enthalten und wird durch den angegebenen Quelloperanden dividiert.

Definierte Kennzeichenbits: keine

Undefinierte Kennzeichenbits: AF, CF, OF, PF, SF, ZF

Beschreibung: IDIV führt eine vorzeichengerechte Division des Akkumulators (und seiner Erweiterung) durch den Quelloperanden durch. Falls der Quelloperand ein Byte ist, wird von dem doppeltlangen Dividenden angenommen, daß er sich in den Registern AL und AH befindet. Der Quotient einfacher Länge wird in AL zurückgegeben, der Rest einfacher Länge in AH. Bei ganzzahliger Division von Bytes ist der größte positive Quotient +127 (7Fh) und der kleinste negative Quotient −127 (81h). Falls der Quelloperand ein Wort ist, wird von dem doppeltlangen Dividenden angenommen, daß er sich in den Registern AX und DX befindet. Der Quotient einfacher Länge wird in AX, der Rest einfacher Länge in DX zurückgegeben. Bei ganzzahliger Division von Wörtern ist der größte positive Quotient +32 767 (7FFFh), und der kleinste negative Quotient ist −32 767 (8001h).

Falls der Quotient positiv ist und das Maximum überschreitet oder negativ und kleiner (d. h. betragsmäßig größer) als das Minimum ist, wird eine Typ-0-Unterbrechung erzeugt. Nichtganzzahlige Quotienten wer-

den abgeschnitten zu ganzen Zahlen (in Richtung 0), und der Rest hat das gleiche Vorzeichen wie der Dividend.

Beispiel:

```
IDIV CL
IDIV BX
```

Codierung:

1111011w	mod 111 r/m

IMUL *Integer Multiply* *Ganzzahlig multiplizieren*

Mnemonische Darstellung: IMUL Quelle

Funktion: Falls der Quelloperand ein Byte ist:
AX = AL x Quelle (mit Vorzeichen)
Falls der Quelloperand ein Wort ist:
AX,DX = AX x Quelle (mit Vorzeichen)

Definierte Kennzeichenbits: CF, OF

Undefinierte Kennzeichenbits: AF, PF, SF, ZF

Beschreibung: IMUL führt eine vorzeichengerechte Multiplikation des Quelloperanden und des Akkumulators aus. Falls der Quelloperand ein Byte ist, wird er multipliziert mit dem Register AL, und das doppeltlange Ergebnis wird in AH und AL zurückgegeben. Falls der Quelloperand ein Wort ist, wird er mit dem Register AX multipliziert und das doppeltlange Ergebnis in die Register DX und AX zurückgegeben. Falls die obere Hälfte des Resultats (AH bei Bytes, DX bei Wörtern) nicht die Vorzeichenerweiterung der unteren Hälfte des Ergebnisses ist, werden CF und OF gesetzt; im anderen Fall werden sie gelöscht. Wenn CF und OF gesetzt sind, zeigen sie an, daß AH oder DX signifikante Stellen des Ergebnisses enthalten.

Beispiele:
```
IMUL BL
IMUL CL
IMUL 400
```

Codierung:

1111011w	mod 101 r/m

IN

Input Byte Or Word
Byte oder Wort von E/A-Anschluß einlesen

Mnemonische Darstellung: IN Akkumulator, Anschluß

Funktion: Falls ein Byte eingelesen werden soll: AL = Daten von Anschlußnummer oder AL = Daten, die durch das DX-Register adressiert werden.
Falls ein Wort eingelesen werden soll: AX = Daten von Anschlußnummer oder AX = Daten, die durch das DX-Register adressiert werden.

Beeinflußte Kennzeichenbits: keine

Beschreibung: IN überträgt ein Byte oder Wort von einem Eingangsanschluß zum AL- bzw. zum AX-Register. Die Anschlußnummer kann entweder durch eine unmittelbar angegebene Bytekonstante (0-255) oder durch eine Zahl, die zuvor in das DX-Register geladen wurde, (0-65535) angegeben werden.

Beispiele:

IN AL, 045H	unmittelbares Byte
IN AX, 046H	unmittelbares Wort
IN AL,DX	Byte bei variablem Anschluß
IN AX,DX	Wort bei variablem Anschluß

Codierung: Fester Anschluß

1110010 w	Anschluß

Variabler Anschluß

1110110 w

Anschlußadresse ist im DX-Register enthalten.

INC

Increment
Inkrementieren

Mnemonische Darstellung: INC Ziel

Funktion: Ziel = Ziel + 1

Definierte Kennzeichenbits: AF, OF, PF, SF, ZF

Beschreibung: INC addiert eine 1 zum Zieloperanden. Der Operand kann ein Byte oder ein Wort sein.

Beispiele:
INC BL
INC BX

Codierung: Speicher oder Registeroperand

1111111w	mod 000 r/m

Registeroperand

01000 reg

INT

Interrupt
Unterbrechen

Mnemonische Darstellung: INT Unterbrechungstyp (0–255)

Funktion: SP = SP − 2; Kennzeichenbits werden auf dem Stapel abgelegt; IF = 0; TF = 0; SP = SP − 2; CS wird auf dem Stapel abgelegt;
Neues CS = Daten von Speicheradresse (Typ * 4 + 2);
SP = SP − 2; IP wird auf dem Stapel abgelegt;
Neues IP = Daten von Speicheradresse Typ * 4.

Definierte Kennzeichenbits: TF, IF

Beschreibung: INT aktiviert die Unterbrechungsbehandlung, die spezifiert wird durch den Unterbrechungstyp, der als Operand angegeben wird. Die Adresse des Unterbrechungszeigers wird durch die Multiplikation des Unterbrechungstyps mit 4 berechnet. Das zweite Wort des Unterbrechungszeigers ersetzt CS. Das erste Wort des Unterbrechungszeigers ersetzt IP.

Falls der Typ als 3 angegeben wurde, erzeugt der Assembler eine spezielle Ein-Byte-Form des Unterbrechungsbefehls. Dies ist die „Breakpoint"-Unterbrechung.

Beispiele:
INT 3
INT 58

Codierung:

1100110 v	Typ, falls v = 1

Falls v = 0, dann Typ = 3

INTO *Interrupt On Overflow*
Bei gesetztem Überlaufskennzeichen unterbrechen

Mnemonische Darstellung: INTO keine Operanden

Funktion: Falls OF = 1 dann: SP = SP − 2; Kennzeichenbits werden auf dem Stapel abgelegt; IF = 0; TF = 0;
SP = SP − 2; CS wird auf dem Stapel abgelegt; Neues CS = Daten der Speicherzelle 12H;
SP = SP − 2; IP wird auf dem Stapel abgelegt; Neues IP = Daten der Speicherzelle 10H.

Beeinflußte Kennzeichenbits: keine

Beschreibung: INTO erzeugt eine Softwareunterbrechung, falls das Überlaufkennzeichenbit (OF) gesetzt ist; im anderen Fall wird die Kontrolle mit dem folgenden Befehl fortgesetzt, ohne die Unterbrechungsbehandlung zu aktivieren.

Codierung:

11001110

IRET *Return From Interrupt*
Rückkehr von Unterbrechung

Mnemonische Darstellung: IRET keine Operanden

Funktion: IP wird vom Stapel geladen; SP = SP + 2; CS wird vom Stapel geladen; SP = SP + 2; Die Kennzeichenbits werden vom Stapel geladen; SP = SP + 2.

Definierte Kennzeichenbits: sämtliche

Beschreibung: IRET überträgt die Kontrolle zurück zum Punkt der Unterbrechung, indem er IP, CS und die Kennzeichenbits vom Stapel lädt. IRET wird zum Verlassen einer Unterbrechungsprozedur benutzt, gleich ob diese durch Hardware oder Software aktiviert wurde.

Codierung:

11001111

JA

Jump On Above
Sprung bei vorzeichenlos größer

JNBE

Jump On Not Below Or Equal
Sprung bei vorzeichenlos nicht kleiner oder gleich

Mnemonische Darstellung:

```
JA    Zielmarke
JNBE  Zielmarke
```

Funktion: Falls CF und ZF=0, dann IP = IP + 8-Bit-Verschiebeanteil der Zielmarke. Der Verschiebeanteil wird auf 16 bit Länge vorzeichenerweitert.

Beeinflußte Kennzeichenbits: keine

Beschreibung: JA und JNBE übertragen, unter der Bedingung daß CF und ZF = 0 sind, die Kontrolle zu dem mit der Zielmarke gekennzeichneten Befehl. Im Objektcode wird allerdings nicht die tatsächliche Adresse dieser Speicherposition codiert, sondern deren Distanz zu dem Befehl, der dem Sprungbefehl folgt. Diese Distanz (displacement, Verschiebeanteil) wird vom Assembler als 8-Bit-Wert im Zweier-Komplement berechnet. Aus diesem Grund kann nur innerhalb eines Bereichs gesprungen werden, der 126 Bytes vor und 127 Bytes nach dem Sprungbefehl umfaßt. Bei der Ausführung des Befehls wird der Verschiebeanteil auf 16 bit Länge vorzeichenerweitert und der Verschiebeanteil zur Berechnung der effektiven Adresse zum laufenden Befehlszeigerwert addiert.

Für beide mnemonischen Darstellungen erzeugt der Assembler denselben Befehlscode.

Beispiele:

```
JA Marke
JNBE Marke
```

Codierung:

01110111	Verschiebeanteil

JAE

Jump On Above Or Equal
Sprung bei vorzeichenlos größer oder gleich

JNB

Jump On Not Below
Sprung bei vorzeichenlos nicht kleiner

Mnemonische Darstellung:

JAE Zielmarke
JNB Zielmarke

Funktion:

Falls CF = 0, dann IP = IP + 8-Bit-Verschiebeanteil der Zielmarke. Der Verschiebeanteil wird auf 16 bit Länge vorzeichenerweitert.

Beeinflußte Kennzeichenbits:

keine

Beschreibung:

JAE und JNB übertragen, falls die Bedingung (CF = 0, d.h. vorzeichenlos größer oder gleich/nicht kleiner) für den getesteten Wert gilt, die Kontrolle zu dem mit der Zielmarke gekennzeichneten Befehl. Im Objektcode wird allerdings nicht die tatsächliche Adresse dieser Speicherposition codiert, sondern deren Distanz zu dem Befehl, der dem Sprungbefehl folgt. Diese Distanz (displacement, Verschiebeanteil) wird vom Assembler als 8-Bit-Wert im Zweier-Komplement berechnet. Aus diesem Grund kann nur innerhalb eines Bereichs gesprungen werden, der 126 Bytes vor und 127 Bytes nach dem Sprungbefehl umfaßt. Bei der Ausführung des Befehls wird der Verschiebeanteil auf 16 bit Länge vorzeichenerweitert und der Verschiebeanteil zur Berechnung der effektiven Adresse zum laufenden Befehlszeigerwert addiert.

Beide mnemonischen Darstellungen erzeugen denselben Befehlscode für den 8086/8088.

Beispiele:

```
JAE  Marke
JNB  Marke
```

Codierung:

01110011	Verschiebeanteil

JB *Jump On Below*
Sprung bei vorzeichenlos kleiner

JNAE *Jump On Not Above Or Equal*
Sprung bei vorzeichenlos nicht größer oder gleich

Mnemonische Darstellung:

```
JB    Zielmarke
JNAE  Zielmarke
```

Funktion: Falls CF = 1, dann IP = IP + 8-Bit-Verschiebeanteil der Zielmarke. Der Verschiebeanteil wird auf 16 bit Länge vorzeichenerweitert.

Beeinflußte Kennzeichenbits: keine

Beschreibung: JB und JNAE übertragen, falls die Bedingung (CF = 1, d.h. vorzeichenlos kleiner/nicht größer oder gleich) für den getesteten Wert gilt, die Kontrolle zu dem mit der Zielmarke gekennzeichneten Befehl. Im Objektcode wird allerdings nicht die tatsächliche Adresse dieser Speicherposition codiert, sondern deren Distanz zu dem Befehl, der dem Sprungbefehl folgt. Diese Distanz (displacement, Verschiebeanteil) wird vom Assembler als 8-Bit-Wert im Zweier-Komplement berechnet. Aus diesem Grund kann nur innerhalb eines Bereichs gesprungen werden, der 126 Bytes vor und 127 Bytes nach dem Sprungbefehl umfaßt. Bei der Ausführung des Befehls wird der Verschiebeanteil auf 16 bit Länge vorzeichenerweitert und der Verschiebeanteil zur Berechnung der effektiven Adresse zum laufenden Befehlszeigerwert addiert.

Beide mnemonischen Darstellungen erzeugen denselben Befehlscode für den 8086/8088.

Beispiele:

```
JB    Marke
JNAE  Marke
```

Codierung:

01110010	Verschiebeanteil

JBE

Jump On Below Or Equal
Sprung bei vorzeichenlos kleiner oder gleich

JNA

Jump On Not Above
Sprung bei vorzeichenlos nicht größer

Mnemonische Darstellung:

JBE Zielmarke
JNA Zielmarke

Funktion:

Falls CF oder ZF = 1, dann IP = IP + 8-Bit- Verschiebeanteil der Zielmarke. Der Verschiebeanteil wird auf 16 bit Länge vorzeichenerweitert.

Beeinflußte Kennzeichenbits:

keine

Beschreibung:

JBE und JNA übertragen, falls die Bedingungen (CF oder ZF = 1, d.h. vorzeichenlos kleiner oder gleich/ oder nicht größer) für den getesteten Wert gelten, die Kontrolle zu dem mit der Zielmarke gekennzeichneten Befehl. Im Objektcode wird allerdings nicht die tatsächliche Adresse dieser Speicherposition codiert, sondern deren Distanz zu dem Befehl, der dem Sprungbefehl folgt. Diese Distanz (displacement, Verschiebeanteil) wird vom Assembler als 8-Bit-Wert im Zweier-Komplement berechnet. Aus diesem Grund kann nur innerhalb eines Bereichs gesprungen werden, der 126 Bytes vor und 127 Bytes nach dem Sprungbefehl umfaßt. Bei der Ausführung des Befehls wird der Verschiebeanteil auf 16 bit Länge vorzeichenerweitert und der Verschiebeanteil zur Berechnung der effektiven Adresse zum laufenden Befehlszeigerwert addiert.

Beide mnemonischen Darstellungen erzeugen denselben Befehl für den 8086/8088.

Beispiele:

JBE Marke
JNA Marke

Codierung:

01110110	Verschiebeanteil

JC

Jump On Carry
Sprung bei gesetztem Übertragskennzeichen

Mnemonische Darstellung: JC Zielmarke

Funktion: Falls CF = 1, dann IP = IP + 8-Bit-Verschiebeanteil der Zielmarke. Der Verschiebeanteil wird auf 16 bit Länge vorzeichenerweitert.

Beeinflußte Kennzeichenbits: keine

Beschreibung: JC überträgt, falls CF = 1, die Kontrolle zu dem mit der Zielmarke gekennzeichneten Befehl. Im Objektcode wird allerdings nicht die tatsächliche Adresse dieser Speicherposition codiert, sondern deren Distanz zu dem Befehl, der dem Sprungbefehl folgt. Diese Distanz (displacement, Verschiebeanteil) wird vom Assembler als 8-Bit-Wert im Zweier-Komplement berechnet. Aus diesem Grund kann nur innerhalb eines Bereichs gesprungen werden, der 126 Bytes vor und 127 Bytes nach dem Sprungbefehl umfaßt. Bei der Ausführung des Befehls wird der Verschiebeanteil auf 16 bit vorzeichenerweitert und der Verschiebeanteil zur Berechnung der effektiven Adresse zum laufenden Befehlszeigerwert addiert.

Beispiel: JC Marke

Codierung:

01110010	Verschiebeanteil

JCXZ *Jump If CX Register Zero*
Sprung, falls CXRegister null

Mnemonische Darstellung: JCXZ Zielmarke

Funktion: Falls CX = 0, dann IP = IP + 8-Bit-Verschiebeanteil der Zielmarke. Der Verschiebeanteil wird auf 16 bit Länge vorzeichenerweitert.

Beeinflußte Kennzeichenbits: keine

Beschreibung: JCXZ überträgt, falls CX = 0 ist, die Kontrolle zu dem mit der Zielmarke gekennzeichneten Befehl. Im Objektcode wird allerdings nicht die tatsächliche Adresse dieser Speicherposition codiert, sondern deren Distanz zu dem Befehl, der dem Sprungbefehl folgt. Diese Distanz (displacement, Verschiebeanteil) wird vom Assembler als 8-Bit-Wert im Zweier-Komplement berechnet. Aus diesem Grund kann nur innerhalb eines Bereichs gesprungen werden, der 126 Bytes vor und 127 Bytes nach dem Sprungbefehl umfaßt. Bei der Ausführung des Befehls wird der Verschiebeanteil auf 16 bit Länge erweitert und der Verschiebeanteil zur Berechnung der effektiven Adresse zum laufenden Befehlszeigerwert addiert.

Dieser Befehl ist nützlich beim Beginn einer Schleife um die Schleife zu umgehen, falls CX den Wert Null hat (d.h. die Schleife nullmal auszuführen).

Beispiel: JCXZ Marke

Codierung:

11100011	Verschiebeanteil

JE *Jump On Equal*
Sprung bei gleich

JZ *Jump On Zero*
Sprung bei Null

Mnemonische Darstellung:
JE Zielmarke
JZ Zielmarke

Funktion:
Falls ZF = 1, dann IP = IP + 8-Bit-Verschiebeanteil der Zielmarke. Der Verschiebeanteil wird auf 16 bit Länge vorzeichenerweitert.

Beeinflußte Kennzeichenbits:
keine

Beschreibung:
JE und JZ übertragen, falls die Bedingung (ZF = 1, d. h. gleich/oder Null) für den getesteten Wert gilt, die Kontrolle zu dem mit der Zielmarke gekennzeichneten Befehl. Im Objektcode wird allerdings nicht die tatsächliche Adresse dieser Speicherposition codiert, sondern deren Distanz zu dem Befehl, der dem Sprungbefehl folgt. Diese Distanz (displacement, Verschiebeanteil) wird vom Assembler als 8-Bit-Wert im Zweier-Komplement berechnet. Aus diesem Grund kann nur innerhalb eines Bereichs gesprungen werden, der 126 vor und 127 Bytes nach dem Sprungbefehl umfaßt. Bei der Ausführung des Befehls wird der Verschiebeanteil auf 16 bit Länge vorzeichenerweitert und der Verschiebeanteil zur Berechnung der effektiven Adresse zum laufenden Befehlszeigerwert addiert.

Beide mnemonischen Darstellungen erzeugen denselben Objektcode für den 8086/8088.

Beispiel:
JE Marke
JZ Marke

Codierung:

01110100	Verschiebeanteil

JG
Jump On Greater Than
Sprung bei größer

JNLE
Jump On Not Less Or Equal
Sprung bei nicht kleiner oder gleich

Mnemonische Darstellung:

```
JG      Zielmarke
JNLE    Zielmarke
```

Funktion: Falls SF = 0F und ZF = 0, dann IP = IP + 8-Bit-Verschiebeanteil der Zielmarke. Der Verschiebeanteil wird auf 16 bit Länge vorzeichenerweitert.

Beeinflußte Kennzeichenbits: keine

Beschreibung: JG und JNLE übertragen, falls die Bedingung (SF ⊻ OF = 0 V ZF = 0, d. h. größer als/nicht kleiner als oder gleich) für den getesteten Wert gilt, die Kontrolle zu dem mit der Zielmarke gekennzeichneten Befehl. Im Objektcode wird allerdings nicht die tatsächliche Adresse dieser Speicherposition codiert, sondern deren Distanz zu dem Befehl, der dem Sprungbefehl folgt. Diese Distanz (displacement, Verschiebeanteil) wird vom Assembler als 8-Bit-Wert im Zweier-Komplement berechnet. Aus diesem Grund kann nur innerhalb eines Bereichs gesprungen werden, der 126 Bytes vor und 127 Bytes nach dem Sprungbefehl umfaßt. Bei der Ausführung des Befehls wird der Verschiebeanteil auf 16 bit Länge vorzeichenerweitert und der Verschiebeanteil zur Berechnung der effektiven Adresse zum laufenden Befehlszeigerwert addiert.

Beide mnemonischen Darstellungen erzeugen denselben Befehl für den 8086/8088.

Beispiele:

```
JG      Marke
JNLE    Marke
```

Codierung:

01111111	Verschiebeanteil

JGE *Jump On Greater Or Equal*
Sprung bei größer oder gleich

JNL *Jump On Not Less*
Sprung bei nicht kleiner

Mnemonische Darstellung:

JGE Zielmarke
JNL Zielmarke

Funktion:

Falls SF = OF, dann IP = IP + 8-Bit-Verschiebeanteil der Zielmarke. Der Verschiebeanteil wird auf 16 bit Länge vorzeichenerweitert.

Beeinflußte Kennzeichenbits: keine

Beschreibung:

JGE und JNL übertragen, falls die Bedingung (SF ⊻ OF = 0, d.h. größer als oder kleiner/nicht kleiner als der getestete Wert) gilt, die Kontrolle zu dem mit der Zielmarke gekennzeichneten Befehl. Im Objektcode wird allerdings nicht die tatsächliche Adresse dieser Speicherposition codiert, sondern deren Distanz zu dem Befehl, der dem Sprungbefehl folgt. Diese Distanz (displacement, Verschiebeanteil) wird vom Assembler als 8-Bit-Wert im Zweier-Komplement berechnet. Aus diesem Grund kann nur innerhalb eines Bereichs gesprungen werden, der 126 Bytes vor und 127 Bytes nach dem Sprungbefehl umfaßt. Bei der Ausführung des Befehls wird der Verschiebeanteil auf 16 bit Länge vorzeichenerweitertund der Verschiebeanteil zur Berechnung der effektiven Adresse zum laufenden Befehlszeigerwert addiert.

Beide mnemonischen Darstellungen erzeugen denselben Befehl für den 8086/8088.

Beispiele:

JGE Marke
JNL Marke

Codierung:

01111101	Verschiebeanteil

JL — *Jump On Less* / *Sprung bei kleiner*

JNGE — *Jump On Not Greater or Equal* / *Sprung bei nicht größer oder gleich*

Mnemonische Darstellung:

JL Zielmarke
JNGE Zielmarke

Funktion:

Falls SF ≠ OF, dann IP = IP + 8-Bit-Verschiebeanteil der Zielmarke. Der Verschiebeanteil wird auf 16 bit Länge vorzeichenerweitert.

Beeinflußte Kennzeichenbits:

keine

Beschreibung:

JL und JNGE übertragen,falls die Bedingung (SF ∨ OF = 1, d.h. kleiner als/nicht größer als oder gleich) für den getesteten Wert gültig ist, die Kontrolle zu dem mit der Zielmarke gekennzeichneten Befehl. Im Objektcode wird allerdings nicht die tatsächliche Adresse dieser Speicherposition codiert, sondern deren Distanz zu dem Befehl, der dem Sprungbefehl folgt. Diese Distanz (displacement, Verschiebeanteil) wird vom Assembler als 8-Bit-Wert im Zweier-Komplement berechnet. Aus diesem Grund kann nur innerhalb eines Bereichs gesprungen werden, der 126 Bytes vor und 127 Bytes nach dem Sprungbefehl umfaßt. Bei der Ausführung des Befehls wird der Verschiebeanteil auf 16 bit Länge vorzeichenerweitert und der Verschiebeanteil zur Berechnung der effektiven Adresse zum laufenden Befehlszeigerwert addiert.

Beide mnemonischen Darstellungen erzeugen denselben Befehl für den 8086/8088.

Beispiele:

JL Marke
JNGE Marke

Codierung:

01111100	Verschiebeanteil

JLE *Jump On Less than or Equal*
Sprung bei kleiner oder gleich

JNG *Jump On Not Greater Than*
Sprung bei nicht größer

Mnemonische Darstellung:

JLE Zielmarke
JNG Zielmarke

Funktion: Falls SF ≠ OF oder ZF = 1, dann IP = IP + 8-Bit-Verschiebeanteil der Zielmarke. Der Verschiebeanteil wird auf 16 bit Länge vorzeichenerweitert.

Beeinflußte Kennzeichenbits: keine

Beschreibung: JLE und JNG übertragen, falls die Bedingung (SF ⊻ OF ∨ ZF = 1, d.h. kleiner als oder gleich/nicht größer als der getestete Wert) gilt, die Kontrolle zu dem mit der Zielmarke gekennzeichneten Befehl. Im Objektcode wird allerdings nicht die tatsächliche Adresse dieser Speicherposition codiert, sondern deren Distanz zu dem Befehl, der dem Sprungbefehl folgt. Diese Distanz (displacement, Verschiebeanteil) wird vom Assembler als 8-Bit-Wert im Zweier-Komplement berechnet. Aus diesem Grund kann nur innerhalb eines Bereichs gesprungen werden, der 126 Bytes vor und 127 Bytes nach dem Sprungbefehl umfaßt. Bei der Ausführung des Befehls wird der Verschiebeanteil auf 16 bit Länge vorzeichenerweitert und der Verschiebeanteil zur Berechnung der effektiven Adresse zum laufenden Befehlszeigerwert addiert.

Beide mnemonischen Darstellungen erzeugen denselben Befehl für den 8086/8088.

Beispiele:

JLE Marke
JNG Marke

Codierung:

01111110	Verschiebeanteil

JMP

Jump Unconditionally
Unbedingter Sprung

Mnemonische Darstellung: JMP Zielmarke

Funktion: Falls intrasegment, dann IP = IP + Verschiebeanteil. Falls intersegment, dann werden CS und IP ersetzt durch die von dem Befehl erhaltenen neuen Werte.

Beeinflußte Kennzeichenbits: keine

Beschreibung: JMP Unconditionally (unbedingter Sprung) überträgt die Kontrolle an den mit der Zielmarke gekennzeichneten Befehl. Der Zieloperand kann von dem Befehl selbst erhalten werden (direkter Sprung) oder von einer von dem Befehl referierten Speicherzelle oder einem Register (indirekter Sprung).

Ein intrasegment-direkter Sprung ändert den Befehlszeiger durch Addition des relativen Abstands des Ziels von dem JMP-Befehl. Falls der Assembler bestimmen kann, daß das Ziel innerhalb von 126 Bytes vor dem JMP-Befehl oder 127 Bytes nach dem JMP-Befehl liegt, erzeugt er automatisch einen aus zwei Bytes bestehenden SHORT-JMP-Befehl (kurzer Sprung). Im anderen Fall adressiert der Assembler ein Ziel innerhalb ± 32K. In jedem dieser Fälle wird die Distanz zum Ziel im Zweier-Komplement im Objektcode codiert und zwar als 8-Bit-Wert (SHORT JMP) bzw. 16-Bit-Wert.

Ein Intrasegment Sprung kann unter Benutzung des Speichers oder eines allgemeinen 16-Bit-Registers durchgeführt werden. Im ersten Fall ersetzt der Inhalt des von dem Befehl referierten Wortes den IP. Im zweiten Fall wird der neue IP-Wert dem in dem Befehl angegebenen Register entnommen.

Ein Intersegment-Sprung ersetzt IP und CS durch in dem Befehl enthaltene Werte. Der indirekte Intersegment-Sprung kann nur über den Speicher erfolgen, wobei das erste Wort des von dem Befehl referierten Doppelwortes IP und das zweite CS ersetzt.

Beispiele: JMP Marke
JMP Marke.Seg

Codierung: Intrasegment direkt

11101001	Versch.-Ant. unt. Byte
	Versch.-Ant. ob. Byte

Intrasegment direkt kurz

11101011	Verschiebeanteil

Intrasegment indirekt

11111111	mod 100 r/m

Intersegment direkt

11101010	Offset unt. Byte	Offset ob. Byte
	Segment unt. Byte	Segment ob. Byte

Intersegment indirekt

11111111	mod 101 r/m

JNC

Jump On Not Carry
Sprung bei nicht gesetztem Übertragskennzeichen

Mnemonische Darstellung: JNC Zielmarke

Funktion: Falls CF = 0, dann IP = IP + 8-Bit-Verschiebeanteil der Zielmarke. Der Verschiebeanteil wird auf 16 bit Länge vorzeichenerweitert.

Beeinflußte Kennzeichenbits: keine

Beschreibung: JNC überträgt bei der Bedingung CF = 0 die Kontrolle zu dem mit der Zielmarke gekennzeichneten Befehl. Im Objektcode wird allerdings nicht die tatsächliche Adresse dieser Speicherposition codiert, sondern deren Distanz zu dem Befehl, der dem Sprungbefehl folgt. Diese Distanz (displacement, Verschiebeanteil) wird vom Assembler als 8-Bit-Wert im Zweier-Komplement berechnet. Aus diesem Grund kann nur innerhalb eines Bereichs gesprungen werden, der 126 Bytes vor und 127 Bytes nach dem Sprungbefehl umfaßt. Bei der Ausführung des Befehls wird der Verschiebeanteil auf 16 bit Länge vorzeichenerweitert und der Verschiebeanteil zur Berechnung der effektiven Adresse zum laufenden Befehlszeigerwert addiert.

Beispiel: JNC Marke

Codierung:

01110011	Verschiebeanteil

JNE *Jump On Not Equal To*
Sprung bei nicht gleich

JNZ *Jump On Not Zero*
Sprung bei nicht Null

Mnemonische Darstellung: JNE Zielmarke

Funktion: Falls ZF = 0, dann IP = IP + 8-Bit-Verschiebeanteil der Zielmarke. Der Verschiebeanteil wird auf 16 bit Länge vorzeichenerweitert.

Beeinflußte Kennzeichenbits: keine

Beschreibung: JNE und JNZ übertragen, falls die getestete Bedingung (ZF = 0) wahr ist, die Kontrolle zu dem mit der Zielmarke gekennzeichneten Befehl. Im Objektcode wird allerdings nicht die tatsächliche Adresse dieser Speicherposition codiert, sondern deren Distanz zu dem Befehl, der dem Sprungbefehl folgt. Diese Distanz (displacement, Verschiebeanteil) wird vom Assembler als 8-Bit-Wert im Zweier-Komplement berechnet. Aus diesem Grund kann nur innerhalb eines Bereichs gesprungen werden, der 126 Bytes vor und 127 Bytes nach dem Sprungbefehl umfaßt. Bei der Ausführung des Befehls wird der Verschiebeanteil auf 16 bit Länge vorzeichenerweitert und der Verschiebeanteil zur Berechnung der effektiven Adresse zum laufenden Befehlszeigerwert addiert.

Beispiel:
```
JNE Marke
JNZ Marke
```

Codierung:

01110101	Verschiebeanteil

JNO

Jump On Not Overflow
Sprung bei nicht gesetztem Überlaufskennzeichen

Mnemonische Darstellung: JNO Zielmarke

Funktion: Falls OF = 0, dann IP = IP + 8-Bit-Verschiebeanteil der Zielmarke. Der Verschiebeanteil wird auf 16 bit Länge vorzeichenerweitert.

Beeinflußte Kennzeichenbits: keine

Beschreibung: JNO überträgt, falls die getestete Bedingung (OF = 0) wahr ist, die Kontrolle zu dem mit der Zielmarke gekennzeichneten Befehl. Im Objektcode wird allerdings nicht die tatsächliche Adresse dieser Speicherposition codiert, sondern deren Distanz zu dem Befehl, der dem Sprungbefehl folgt. Diese Distanz (displacement, Verschiebeanteil) wird vom Assembler als 8-Bit-Wert im Zweier-Komplement berechnet. Aus diesem Grund kann nur innerhalb eines Bereichs gesprungen werden, der 126 Bytes vor und 127 Bytes nach dem Sprungbefehl umfaßt. Bei der Ausführung des Befehls wird der Verschiebeanteil auf 16 bit Länge vorzeichenerweitert und der Verschiebeanteil zur Berechnung der effektiven Adresse zum laufenden Befehlszeigerwert addiert.

Beispiel: JNO Marke

Codierung:

01110001	Verschiebeanteil

JNS

Jump On Not Sign
Sprung bei nicht gesetztem Vorzeichenkennzeichen

Mnemonische Darstellung: JNS Zielmarke

Funktion: Falls SF = 0, dann IP = IP + 8-Bit-Verschiebeanteil der Zielmarke. Der Verschiebeanteil wird auf 16 bit Länge vorzeichenerweitert.

Beeinflußte Kennzeichenbits: keine

Beschreibung: JNS überträgt, falls die getestete Bedingung (SF = 0) wahr ist, die Kontrolle zu dem mit der Zielmarke gekennzeichneten Befehl. Im Objektcode wird allerdings nicht die tatsächliche Adresse dieser Speicherposition codiert, sondern deren Distanz zu dem Befehl, der dem Sprungbefehl folgt. Diese Distanz (displacement, Verschiebeanteil) wird vom Assembler als 8-Bit-Wert im Zweier-Komplement berechnet. Aus diesem Grund kann nur innerhalb eines Bereichs gesprungen werden, der 126 Bytes vor und 127 Bytes nach dem Sprungbefehl umfaßt. Bei der Ausführung des Befehls wird der Verschiebeanteil auf 16 bit Länge vorzeichenerweitert und der Verschiebeanteil zur Berechnung der effektiven Adresse zum laufenden Befehlszeigerwert addiert.

Beispiel: JNS Marke

Codierung:

01111001	Verschiebeanteil

JNP

Jump On Not Parity
Sprung bei nicht gesetztem Paritätskennzeichen

JPO

Jump On Parity Odd
Sprung bei ungerader Parität

Mnemonische Darstellung:

JNP Zielmarke
JPO Zielmarke

Funktion:

Falls PF = 0, dann IP = IP + 8-Bit-Verschiebeanteil der Zielmarke. Der Verschiebeanteil wird auf 16 bit Länge vorzeichenerweitert.

Beeinflußte Kennzeichenbits:

keine

Beschreibung:

JNP und JPO übertragen, falls die getestete Bedingung (PF = 0) wahr ist, die Kontrolle zu dem mit der Zielmarke gekennzeichneten Befehl. Im Objektcode wird allerdings nicht die tatsächliche Adresse dieser Speicherposition codiert, sondern deren Distanz zu dem Befehl, der dem Sprungbefehl folgt. Diese Distanz (displacement, Verschiebeanteil) wird vom Assembler als 8-Bit-Wert im Zweier-Komplement berechnet. Aus diesem Grund kann nur innerhalb eines Bereichs gesprungen werden, der 126 Bytes vor und 127 Bytes nach dem Sprungbefehl umfaßt. Bei der Ausführung des Befehls wird der Verschiebeanteil auf 16 bit Länge vorzeichenerweitert und der Verschiebeanteil zur Berechnung der effektiven Adresse zum laufenden Befehlszeigerwert addiert.

Beide mnemonischen Darstellungen erzeugen denselben Befehl für den 8086/8088.

Beispiele:

```
JNP  Marke
JPO  Marke
```

Codierung:

01111011	Verschiebeanteil

JO

Jump On Overflow
Sprung bei gesetztem Überlaufskennzeichen

Mnemonische Darstellung: JO Zielmarke

Funktion: Falls OF = 1, dann IP = IP + 8-Bit-Verschiebeanteil der Zielmarke. Der Verschiebeanteil wird auf 16 bit Länge vorzeichenerweitert.

Beeinflußte Kennzeichenbits: keine

Beschreibung: JO überträgt, falls die getestete Bedingung (OF = 1) wahr ist, die Kontrolle zu dem mit der Zielmarke gekennzeichneten Befehl. Im Objektcode wird allerdings nicht die tatsächliche Adresse dieser Speicherposition codiert, sondern deren Distanz zu dem Befehl, der dem Sprungbefehl folgt. Diese Distanz (displacement, Verschiebeanteil) wird vom Assembler als 8-Bit-Wert im Zweier-Komplement berechnet. Aus diesem Grund kann nur innerhalb eines Bereichs gesprungen werden, der 126 Bytes vor und 127 Bytes nach dem Sprungbefehl umfaßt. Bei der Ausführung des Befehls wird der Verschiebeanteil auf 16 bit Länge vorzeichenerweitert und der Verschiebeanteil zur Berechnung der effektiven Adresse zum laufenden Befehlszeigerwert addiert.

Beispiel: JO Marke

Codierung:

01110000	Verschiebeanteil

JP

Jump On Parity
Sprung bei gesetztem Paritätskennzeichen

JPE

Jump On Parity Even
Sprung bei gerader Parität

Mnemonische Darstellung:

JP Zielmarke
JPE Zielmarke

Funktion:

Falls PF = 1, dann IP = IP + 8-Bit-Verschiebeanteil der Zielmarke. Der Verschiebeanteil wird auf 16 bit Länge vorzeichenerweitert.

Beeinflußte Kennzeichenbits:

keine

Beschreibung:

JP und JPE übertragen, falls die getestete Bedingung (PF = 1) wahr ist, die Kontrolle zu dem mit der Zielmarke gekennzeichneten Befehl. Im Objektcode wird allerdings nicht die tatsächliche Adresse dieser Speicherposition codiert, sondern deren Distanz zu dem Befehl, der dem Sprungbefehl folgt. Diese Distanz (displacement, Verschiebeanteil) wird vom Assembler als 8-Bit-Wert im Zweier-Komplement berechnet. Aus diesem Grund kann nur innerhalb eines Bereichs gesprungen werden, der 126 Bytes vor und 127 Bytes nach dem Sprungbefehl umfaßt. Bei der Ausführung des Befehls wird der Verschiebeanteil auf 16 bit Länge vorzeichenerweitert und der Verschiebeanteil zur Berechnung der effektiven Adresse zum laufenden Befehlszeigerwert addiert.

Beispiele:

```
JP    Marke
JPE   Marke
```

Codierung:

01111010	Verschiebeanteil

JS

Jump On Sign
Sprung bei gesetztem Vorzeichenkennzeichen

Mnemonische Darstellung: JS Zielmarke

Funktion: Falls SF = 1, dann IP = IP + 8-Bit-Verschiebeanteil der Zielmarke. Der Verschiebeanteil wird auf 16 bit Länge vorzeichenerweitert.

Beeinflußte Kennzeichenbits: keine

Beschreibung: JS überträgt, falls die getestete Bedingung (SF = 1) wahr ist, die Kontrolle zu dem mit der Zielmarke gekennzeichneten Befehl. Im Objektcode wird allerdings nicht die tatsächliche Adresse dieser Speicherposition codiert, sondern deren Distanz zu dem Befehl, der dem Sprungbefehl folgt. Diese Distanz (displacement, Verschiebeanteil) wird vom Assembler als 8-Bit-Wert im Zweier-Komplement berechnet. Aus diesem Grund kann nur innerhalb eines Bereichs gesprungen werden, der 126 Bytes vor und 127 Bytes nach dem Sprungbefehl umfaßt. Bei der Ausführung des Befehls wird der Verschiebeanteil auf 16 bit Länge vorzeichenerweitert und der Verschiebeanteil zur Berechnung der effektiven Adresse zum laufenden Befehlszeigerwert addiert.

Beispiel: JS Marke

Codierung:

01111000	Verschiebeanteil

LAHF *Load Register AH From Flags*
Register AH mit Kennzeichenbits laden

Mnemonische Darstellung: LAHF keine Operanden

Funktion: AH = d7 d6 d5 d4 d3 d2 d1 d0
SF ZF X AF X PF X CF, wobei X = undefiniert

Beeinflußte Kennzeichenbits: keine

Beschreibung: LAHF (load register AH from flags, lade Register AH mit den Kennzeichenbits) kopiert SF,ZF,AF,PF,CF in die Bits 7,6,4,2,0 des AH-Registers. Die Inhalte der Bits 5,3 und 1 sind undefiniert.

Codierung: 10011111

LDS

Load Pointer Using DS
Zeiger einschließlich DS laden

Mnemonische Darstellung: LDS Ziel, Quelle

Funktion: REG = Daten vom Quelloperanden
DS = Daten vom (Quelloperanden + 2)

Beeinflußte Kennzeichenbits: keine

Beschreibung: LDS überträgt eine 32-Bit-Zeigervariable vom Quelloperanden, der ein Speicheroperand sein muß, zum Zieloperanden und zum Register DS. Das Offsetwort des Zeigers wird zum Zieloperanden übertragen, welcher irgendein allgemeines 16-Bit-Register sein kann. Das Segmentwort des Zeigers wird zum DS-Register übertragen.

Beispiel: LDS SI, Marke

Codierung:

11000101	mod reg r/m

LEA

Load Effective Address
Effektive Adresse laden

Mnemonische Darstellung: LEA Ziel, Quelle

Funktion: REG = Offset des Quelloperanden

Beeinflußte Kennzeichenbits: keine

Beschreibung: LEA überträgt das Offset des Quelloperanden (und nicht seinen Wert) zum Zieloperanden. Der Quelloperand muß ein Speicheroperand sein, und der Zieloperand muß ein allgemeines 16-Bit-Register sein.

Beispiel: LEA CX,Marke

Codierung:

10001101	mod reg r/m

LES

Load Pointer Using ES
Zeiger einschließlich ES laden

Mnemonische Darstellung: LES Ziel, Quelle

Funktion: REG = Daten von Quelladresse
ES = Daten von (Quelladresse + 2)

Beeinflußte Kennzeichenbits: keine

Beschreibung: LES überträgt eine 32-Bit-Zeigervariable vom Quelloperanden, der ein Speicheroperand sein muß, zum Zieloperanden und zum Register ES. Das Offsetwort des Zeigers wird zum Zieloperanden übertragen, welcher irgendein allgemeines 16-Bit-Register sein kann. Das Segmentwort des Zeigers wird in das Register ES übertragen.

Beispiel:

```
LES DI,Marke
Marke        0215H
Marke + 2    0457H

Vorher:      Nachher:
DI = 0158H   DI = 0215H
ES = 1005H   ES = 0457 H
```

Codierung:

11000100	mod reg r/m

LOCK *Lock The Bus*
LOCK-Signal auf dem Bus durchsetzen

Mnemonische Darstellung: LOCK MOV AX, Marke

Funktion: keine

Beeinflußte Kennzeichenbits: keine

Beschreibung: LOCK ist ein Ein-Byte-Präfix, das bewirkt, daß der 8088 (konfiguriert im Maximum-Modus) während der folgenden Befehlsabarbeitung das LOCK-Bussignal durchsetzt.

Codierung:

11110000

LODS *Load String (Byte Or Word)*
Zeichenkette (Byte oder Wort) laden

Mnemonische Darstellung: LODS Quellzeichenkette

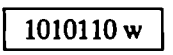
Funktion: Falls Byte-Zeichenkette : AL = Daten aus dem Speicher, adressiert von SI:
Falls DF = 0, dann SI = SI + 1
Falls DF = 1, dann SI = SI − 1

Falls Wort-Zeichenkette : AX = Daten aus dem Speicher, adressiert von SI:
Falls DF = 0, dann SI = SI + 2
Falls DF = 1, dann SI = SI − 2

Beeinflußte Kennzeichenbits: keine

Beschreibung: LODS überträgt das Byte- oder Wort-Zeichenkettenelement, das von SI adressiert wird, ins Register AL bzw. AX und aktualisiert SI so, daß es zum nächsten Element der Zeichenkette zeigt. Der Assembler benutzt die Quellzeichenkette, die in dem Befehl angegeben wird, um den Typ und die Zugreifbarkeit der Daten zu bestimmen.

Codierung:



LOOP *Loop If CX Not Zero*
Schleife, falls CX nicht Null

Mnemonische Darstellung: LOOP Zielmarke

Funktion: CX = CX − 1: Falls CX ungleich Null ist, dann IP = IP + 8-Bit-Verschiebeanteil der Zielmarke. Der Verschiebeanteil wird auf 16 bit Länge vorzeichenerweitert. Falls CX = 0 ist, wird der nächste Befehl, der auf LOOP folgt, ausgeführt.

Beeinflußte Kennzeichenbits: keine

Beschreibung: LOOP erniedrigt CX um 1 und überträgt, falls CX nicht 0 ist, die Kontrolle zu dem mit der Zielmarke gekennzeichneten Befehl. Im Objektcode wird allerdings nicht die tatsächliche Adresse dieser Speicherposition codiert, sondern deren Distanz zu dem Befehl, der dem Sprungbefehl folgt. Diese Distanz (displacement, Verschiebeanteil) wird vom Assembler als 8-Bit-Wert im Zweier-Komplement berechnet. Aus diesem Grund kann nur innerhalb eines Bereichs gesprungen werden, der 126 Bytes vor und 127 Bytes nach dem Sprungbefehl umfaßt. Bei der Ausführung des Befehls wird der Verschiebeanteil auf 16 bit Länge vorzeichenerweitert und der Verschiebeanteil zur Berechnung der effektiven Adresse zum laufenden Befehlszeigerwert addiert. Anderenfalls wird der auf LOOP folgende Befehl ausgeführt.

Codierung:

11100010	Verschiebeanteil

LOOPE *Loop While Equal*
Schleife, solange gleich

LOOPZ *Loop While Zero*
Schleife, solange Null

Mnemonische Darstellung:
LOOPE Zielmarke
LOOPZ Zielmarke

Funktion:
CX = CX − 1: Falls CX ungleich Null und ZF = 1 ist, dann IP = IP + 8-Bit-Verschiebeanteil der Zielmarke. Der Verschiebeanteil wird auf 16 bit Länge vorzeichenerweitert.

Beeinflußte Kennzeichenbits:
keine

Beschreibung:
LOOPE und LOOPZ sind verschiedene mnemonische Darstellungen für denselben Befehl. CX wird um 1 erniedrigt, und falls CX nicht 0 und ZF = 1 ist, wird die Kontrolle zu dem mit der Zielmarke gekennzeichneten Befehl übertragen. Im Objektcode wird allerdings nicht die tatsächliche Adresse dieser Speicherposition codiert, sondern deren Distanz zu dem Befehl, der dem Sprungbefehl folgt. Diese Distanz (displacement, Verschiebeanteil) wird vom Assembler als 8-Bit-Wert im Zweier-Komplement berechnet. Aus diesem Grund kann nur innerhalb eines Bereichs gesprungen werden, der 126 Bytes vor und 127 Bytes nach dem Sprungbefehl umfaßt. Bei der Ausführung des Befehls wird der Verschiebeanteil auf 16 bit Länge vorzeichenerweitert und der Verschiebeanteil zur Berechnung der effektiven Adresse zum laufenden Befehlszeigerwert addiert. Anderenfalls wird der auf LOOPE/LOOPZ folgende Befehl ausgeführt.

Beispiele:
LOOPE Marke
LOOPZ Marke

Codierung:

11100001	Verschiebeanteil

LOOPNZ *Loop While not Zero*
Schleife, solange nicht Null

LOOPNE *Loop While Not Equal*
Schleife, solange nicht gleich

Mnemonische Darstellung:
LOOPNZ Zielmarke
LOOPNE Zielmarke

Funktion:
CX = CX − 1: Falls CX ungleich Null und ZF = 0 ist, dann IP = IP + 8-Bit-Verschiebeanteil der Zielmarke. Der Verschiebeanteil wird auf 16 bit Länge vorzeichenerweitert.

Beeinflußte Kennzeichenbits:
keine

Beschreibung:
LOOPNZ und LOOPNE sind verschiedene mnemonische Darstellungen für denselben Befehl. CX wird um 1 erniedrigt, und falls CX nicht 0 und ZF = 0 ist, wird die Kontrolle zu dem mit der Zielmarke gekennzeichneten Befehl übergeben. Im Objektcode wird allerdings nicht die tatsächliche Adresse dieser Speicherposition codiert, sondern deren Distanz zu dem Befehl, der dem Sprungbefehl folgt. Diese Distanz (displacement, Verschiebeanteil) wird vom Assembler als 8-Bit-Wert im Zweier-Komplement berechnet. Aus diesem Grund kann nur innerhalb eines Bereichs gesprungen werden, der 126 Bytes vor und 127 Bytes nach dem Sprungbefehl umfaßt. Bei der Ausführung des Befehls wird der Verschiebeanteil auf 16 bit Länge vorzeichenerweitert und der Verschiebeanteil zur Berechnung der effektiven Adresse zum laufenden Befehlszeigerwert addiert. Anderenfalls wird der auf LOOPNE/LOOPNZ folgende Befehl ausgeführt.

Beispiele:
LOOPNZ Marke
LOOPNE Marke

Codierung:

11100000	Verschiebeanteil

MOV *Move Byte or Word*
Byte oder Wort übertragen

Mnemonische Darstellung: MOV Ziel, Quelle

Funktion: Ziel = Quelle

Beeinflußte Kennzeichenbits: keine

Beschreibung: MOV überträgt ein Byte oder ein Wort vom Quelloperanden zum Zieloperanden.

Beispiele:
```
MOV AX,BX
MOV CL,AL
MOV BX, 300
MOV BH, 25
```

Codierung: Speicher- oder Registeroperand zu/von Registeroperand

100010dw	mod reg r/m

Unmittelbarer Operand zu Speicher- oder Registeroperand

1100011w	mod 000 r/m	Daten	Daten, falls w = 1

Unmittelbarer Operand zu Register

1011w reg	Daten	Daten, falls w = 1

Speicheroperand zu Akkumulator

1010000w	unt. Adr.-Byte	ob. Adr.-Byte

Akkumulator zu Speicheroperand

1010001w	unt. Adr.-Byte	ob. Adr.-Byte

Speicher- oder Registeroperand zu Segmentregister

10001110	mod 0 reg r/m

Segmentregister zu Speicher- oder Registeroperand

10001100	mod 0 reg r/m

MOVS *Move String* *Zeichenkette übertragen*

Mnemonische Darstellung: MOVS Ziel-Zeichenkette, Quell-Zeichenkette

Funktion: Ziel-Zeichenkette = Quell-Zeichenkette

Beeinflußte Kennzeichenbits: keine

Beschreibung: MOVS überträgt ein Byte oder ein Wort von der Quell- Zeichenkette (adressiert durch SI) zur Ziel-Zeichenkette (adressiert durch DI) und aktualisiert SI und DI, so daß diese auf das nächste Zeichenkettenelement zeigen. Wenn der Befehl MOVS mit REP benutzt wird, führt er eine Speicher-zu-Speicher Blockübertragung durch. Die in dem Befehl spezifizierten Zeichenkettenoperanden für Quelle und Ziel werden vom Assembler benutzt, um Typ und Zugreifbarkeit der Operanden zu bestimmen.

Beispiel: MOVS Puffer1, Puffer2

Codierung:

1010010w

MUL *Multiply*
Multiplizieren

Mnemonische Darstellung: MUL Quelle

Funktion: Falls der Quelloperand ein Byte ist, dann AX = AL x Quelle (vorzeichenlos).

Falls der Quelloperand ein Wort ist, dann AX,DX = AX x Quelle (vorzeichenlos).

Definierte Kennzeichenbits: CF,OF

Undefinierte Kennzeichenbits: AF,PF,SF,ZF

Beschreibung: MUL führt eine vorzeichenlose Multiplikation des Quelloperanden und des Akkumulators durch. Falls der Quelloperand ein Byte ist, wird er mit dem Register AL multipliziert, und das doppeltlange Ergebnis wird in AH und AL zurückgespeichert. Falls der Quelloperand ein Wort ist, wird er mit dem Register AX multipliziert, und das doppeltlange Ergebnis wird in die Register DX und AX geschrieben. Die Operanden werden wie vorzeichenlose binäre Zahlen behandelt. Falls die obere Hälfte des Ergebnisses (AH bei Byteoperanden, DX bei Wortoperanden) nicht gleich Null ist, werden CF und OF gesetzt; anderenfalls werden sie gelöscht. Falls CF und OF gesetzt sind, zeigen sie an, daß AH oder DX signifikante Stellen des Ergebnisses enthalten.

Beispiele:
```
MUL 25
MUL CX
MUL BL
```

Codierung:

1111011w	mod 100 r/m

NEG

Negate (Form Two's Complement)
Negieren (Zweier-Komplement bilden)

Mnemonische Darstellung: NEG Ziel

Funktion: Ziel = 0 − Ziel

Definierte Kennzeichenbits: AF,CF,OF,PF,SF,ZF

Beschreibung: NEG subtrahiert den Zieloperanden, welcher ein Byte oder ein Wort sein kann, von 0 und bringt das Resultat zurück in den Zieloperanden. Es wird dadurch das Zweier-Komplement der Zahl gebildet.

Codierung:

1111011w	mod 011 r/m

NOP *No Operation*
Keine Operation

Mnemonische Darstellung: NOP

Funktion: keine

Beeinflußte Kennzeichenbits: keine

Beschreibung: NOP bewirkt keine Operation der CPU.

Codierung: 10010000

NOT *Logical Not*
Logisch invertieren

Mnemonische Darstellung: NOT Ziel

Funktion: Ziel = Einer-Komplement des Ziels

Beeinflußte Kennzeichenbits: keine

Beschreibung: NOT invertiert die Bits (bildet das Einer-Komplement) des Bytes oder des Wortes des Zieloperanden.

Beispiel: NOT AL

Codierung:

1111011w	mod 010 r/m

OR

Logical Or
Logisches Oder bilden

Mnemonische Darstellung: OR Ziel, Quelle

Funktion: Ziel = Ziel V Quelle : CF = 0: OF = 0

Definierte Kennzeichenbits: CF,OF,PF,SF,ZF

Undefinierte Kennzeichenbits: AF

Beschreibung: OR führt das logische „Oder" der beiden Operanden (Byte oder Wort) durch und gibt das Ergebnis in den Zieloperanden zurück. Ein Bit in dem Ergebnis wird gesetzt, falls ein oder beide der korrespondierenden Bits in den Ausgangsoperanden gesetzt sind; anderenfalls wird das Ergebnisbit gelöscht.

Beispiele:
OR AL,BL
OR AL, 00111010B

Codierung: Speicher- oder Registeroperand mit Registeroperand

000010dw	mod reg r/m

Unmittelbarer Operand zu Speicher- oder Registeroperand

1000000w	mod 001 r/m	Daten	Daten, falls w = 1

Unmittelbarer Operand zum Akkumulator

0000110w	Daten	Daten, falls w = 1

OUT *Output* *Ausgeben*

Mnemonische Darstellung: OUT Anschluß, Akkumulator

Funktion: Falls AL angegeben ist, dann wird AL zum Anschluß ausgegeben.
Falls AX angegeben wird, dann wird AX zu Anschluß und (Anschluß +1) ausgegeben.

Beeinflußte Kennzeichenbits: keine

Beschreibung: OUT überträgt ein Byte oder Wort vom AL- oder AX-Register zu einem Ausgabeanschluß. Die Anschlußnummer kann entweder durch eine unmittelbare Bytekonstante angegeben werden, dies erlaubt Zugriff auf die Anschlüsse 0–255, oder durch eine Zahl, die zuvor in das DX-Register gebracht wurde, wodurch ein variabler Zugriff zu den Anschlüssen 0–65535 erlaubt wird.

Beispiele:
```
OUT 45,AX
OUT DX,AL
```

Codierung: Fester Anschluß

1110011w	Anschluß

Variabler Anschluß: DX ist die Adresse des Anschlusses.

1110111w

POP

Pop Stack
Vom Stapel entnehmen

Mnemonische Darstellung: POP Ziel

Funktion: Ziel = Daten auf Stapelspitze; SP = SP + 2

Beeinflußte Kennzeichenbits: keine

Beschreibung: POP überträgt das aktuelle oberste Wort des Stapels (auf das SP zeigt) zum Zieloperanden und erhöht dann SP um 2. SP zeigt damit auf die neue Stapelspitze.

Beispiele:
POP DX
POP DS CS darf nicht bei POP angegeben werden

Codierung: Speicher- oder Registeroperand

10001111	mod 000 r/m

Registeroperand

01011 reg

Segmentregister

000 reg 111

POPF *Pop Flags*
Kennzeichen vom Stapel laden

Mnemonische Darstellung: POPF

Funktion: Die Kennzeichenbits werden durch die Daten der Stapelspitze ersetzt; SP = SP + 2.

Definierte Kennzeichenbits: sämtliche

Beschreibung: POPF überträgt bestimmte Bits vom Wort an der aktuellen Stapelspitze (auf die das Register SP zeigt) in die Kennzeichenbits und überschreibt so alle vorher dort enthaltenen Werte. SP wird danach um 2 erhöht.

POPF kann auch benutzt werden, um den Wert von TF zu setzen, wofür es keinen eigenen Befehl gibt.

Codierung:

10011100

PUSH *Push Data Of Top Of Stack*
Daten auf dem Stapel ablegen

Mnemonische Darstellung: PUSH Quelle

Funktion: SP = SP − 2: der Quelloperand wird auf dem Stapel abgelegt.

Beeinflußte Kennzeichenbits: keine

Beschreibung: PUSH erniedrigt SP (den Stapelzeiger) um 2 und überträgt dann ein Wort vom Quelloperanden zur neuen Stapelspitze, auf die SP nun zeigt.

Beispiele:
PUSH SI
PUSH ES CS darf auch auf dem Stapel abgelegt werden

Codierung: Speicher- oder Registeroperand

11111111	mod 110 r/m

Registeroperand

01010 reg

Segmentregister

000 reg 110

PUSHF *Push Flags*
Kennzeichenbits auf dem Stapel ablegen

Mnemonische Darstellung: PUSHF

Funktion: SP = SP − 2; Das Register der Kennzeichenbits wird auf dem Stapel abgelegt.

Beeinflußte Kennzeichenbits: keine

Beschreibung: PUSHF erniedrigt SP (den Stapelzeiger) um 2 und überträgt sämtliche Kennzeichenbits zum Wort der Stapelspitze, auf das SP zeigt. Die Kennzeichenbits werden nicht beeinflußt.

Codierung:

10011101

RCL

Rotate Through Carry Left
Durch Übertrag links rotieren

Mnemonische Darstellung: RCL Quelle, Anzahl

Funktion: temp = Anzahl: Falls temp ungleich Null ist, dann:
temcary = CF; CF = oberes Bit des Ziels;
Ziel = (Ziel x 2) + temcary;
temp = temp − 1; wiederhole Funktion bis temp = 0.
Falls Anzahl = 1, dann: Falls oberes Bit des Ziels ungleich zu CF ist, dann OF = 1; sonst OF = 0.
Falls Anzahl nicht gleich 1 ist, ist OF undefiniert.

Definierte Kennzeichenbits: CF,OF

Beschreibung: RCL rotiert die Bits in dem Byte oder Wort des Zieloperanden um die in dem Operanden „Anzahl" angegebene Zahl von Bits nach links. Das Übertragsbit (CF, carry flag) wird als „Teil des Zieloperanden behandelt; das bedeutet, daß sein Wert in das untere Bit des Ziels rotiert und mit dem oberen Bit des Ziels überschrieben wird.

Beispiele:

```
MOV CL, 3
RCL AL,CL
```

Vorher:	Nachher:
AL = 01011110, CF = 0	AL = 10111100, CF = 0 Anz. 1
AL = 10111100, CF = 0	AL = 01111000, CF = 1 Anz. 2
AL = 01111000, CF = 1	AL = 11110001, CF = 0 Anz. 3

Codierung:

110100vw	mod 010 r/m

Falls v = 0, dann Anzahl = 1

RCR *Rotate Through Carry Right*
Durch Übertrag rechts rotieren

Mnemonische Darstellung: RCR Ziel, Anzahl

Funktion: temp = Anzahl: Falls temp ungleich 0 ist, dann: temcary = CF; CF = niederes Bit des Ziels;
Ziel = Ziel / 2;
oberes Bit des Ziels = temcary; temp = temp − 1; wiederhole Funktion bis temp = 0.
Falls Anzahl = 1, dann: Falls oberes Bit des Ziels nicht gleich dem nächstoberen Bit des Ziels ist, dann OF = 1, anderenfalls OF = 0.
Falls Anzahl ungleich 1 ist, ist OF undefiniert.

Definierte Kennzeichenbits: CF,OF

Beschreibung: RCR rotiert die Bits in dem Byte oder Wort des Zieloperanden, um die in dem Operanden „Anzahl" angegebene Zahl von Bits nach rechts. Das Übertrags-Kennzeichenbit (CF, carry flag) wird als „Teil" des Zieloperanden behandelt; das bedeutet, daß sein Wert in das obere Bit des Ziels rotiert und durch das untere Bit des Ziels überschrieben wird.

Beispiele:

```
MOV CL, 2
RCL BL,CL
```

Vorher:	Nachher:
BL = 11000010, CF = 1	BL = 11100001, CF = 0 Anz. 1
BL = 11100001, CF = 0	BL = 01110000, CF = 1 Anz. 2

Codierung:

110100vw	mod 011 r/m

Falls v = 0, dann Anzahl = 1

REP *Repeat*
Wiederholen

REPE *Repeat While Equal*
Wiederholen, solange gleich

REPZ *Repeat While Zero*
Wiederholen, solange Null

Mnemonische Darstellung: REP MOVS Ziel, Quelle

Beeinflußte Kennzeichenbits: keine

Beschreibung: REP wird in Verbindung mit dem MOVS-Befehl (MOVE STRING, Übertrage Zeichenkette) und dem STOS-Befehl (STORE STRING, Speichere Zeichenkette) benutzt und wird als „wiederhole, solange nicht Ende der Zeichenkette erreicht" (CX ungleich 0) interpretiert.

REPE und REPZ haben dieselbe Funktion und haben auch physikalisch dasselbe Präfixbyte wie REP. Diese Befehle werden mit dem CMPS-Befehl (COMPARE STRING, Vergleiche Zeichenketten) und SCAS-Befehl (SCAN STRING, Suche Zeichenkettenelement) benutzt und erfordern, daß das ZF-Kennzeichenbit (das von diesen Befehlen verändert wird) vor der Durchführung der nächsten Wiederholung gesetzt wird. ZF muß nicht vor der Durchführung des wiederholten Zeichenkettenbefehls initialisiert werden.

Wiederholte Zeichenkettenbefehle sind unterbrechbar; der Prozessor erkennt das Vorliegen einer Unterbrechungsanforderung vor der Verarbeitung des nächsten Zeichenkettenelements. Die Verarbeitung von Systemunterbrechungen wird in keiner Weise beeinflußt. Bei der Rückkehr von der Unterbrechung wird die wiederholte Operation beim Punkt der Unterbrechung fortgesetzt.

Beispiele:
```
REP   MOVS Ziel, Quelle
REPE  CMPS Ziel, Quelle
```

Codierung:

1111001z

REPNE *Repeat While Not Equal*
Wiederholen, solange nicht gleich

REPNZ *Repeat While Not Zero*
Wiederholen, solange nicht Null

Mnemonische Darstellung: REPNE SCAS Ziel

Beeinflußte Kennzeichenbits: keine

Beschreibung: REPNE und REPNZ haben dieselbe Funktion und haben auch physikalisch dasselbe Präfixbyte wie REP. Diese Befehle werden mit dem CMPS-Befehl (COMPARE STRING, Vergleiche Zeichenketten) und SCAS-Befehl (SCAN STRING, Suche Zeichenkettenelement) benutzt. ZF muß gelöscht werden, oder die Wiederholung wird beendet. ZF muß vor dem Beginn des wiederholten Zeichenkettenbefehls nicht initialisiert werden.

Wiederholte Zeichenkettenbefehle sind unterbrechbar; der Prozessor erkennt das Vorliegen einer Unterbrechungsanforderung vor der Verarbeitung des nächsten Zeichenkettenelements. Die Verarbeitung von Systemunterbrechungen wird in keiner Weise beeinflußt. Bei der Rückkehr von der Unterbrechung wird die wiederholte Operation beim Punkt der Unterbrechung fortgesetzt.

Beispiel: REPNE SCAS Ziel

Codierung:

1111001z

RET

Return
Von Prozedur zurückkehren

Mnemonische Darstellung: RET optionaler Stapelkorrekturwert

Funktion: Falls intrasegment: IP wird vom Stapel geladen; SP = SP + 2.
Falls intersegment: CS wird vom Stapel geladen; SP = SP + 2. IP wird vom Stapel geladen; SP = SP + 2.
Falls der optionale Stapelkorrekturwert angegeben ist, dann SP = SP + Stapelkorrekturwert.

Beeinflußte Kennzeichenbits: keine

Beschreibung: RET überträgt die Kontrolle von einer Prozedur zurück an den Befehl, der dem CALL folgt, das die Prozedur aktiviert hat. Der Assembler erzeugt ein Intrasegment-RET, falls der Programmierer die Prozedur als NEAR definiert hatte, oder ein Intersegment-RET, falls die Prozedur als FAR definiert wurde. RET übernimmt das Wort an der Stapelspitze (auf das SP zeigt) in den Befehlszeiger und erhöht SP um 2. Falls ein Intersegment-RET vorliegt, wird das neue Wort an der Stapelspitze in das CS-Register übernommen und SP um 2 erhöht. Falls ein Stapelkorrekturwert angegeben wurde, addiert RET diesen Wert zu SP. Diese Befehlsvariante kann benutzt werden, um Parameter freizugeben, die vor dem Aufruf des CALL-Befehls auf dem Stapel abgelegt wurden.

Beispiele:
```
RET
RET 6
```

Codierung: Intrasegment

11000011

Intrasegment und Addition eines Korrekturwerts zu SP

11000010	Daten unt. Byte	Daten ob. Byte

Intersegment

11001011

Intersegment und Addition eines Korrekturwerts zu SP

11001010	Daten unt. Byte	Daten ob. Byte

ROL *Rotate Left* *Links rotieren*

Mnemonische Darstellung: ROL Ziel, Anzahl

Funktion: temp = Anzahl; Falls temp ungleich Null ist, dann: CF = oberes Bit des Ziels;
Ziel = Ziel x 2 + CF; temp = temp − 1; wiederhole Funktion bis temp = 0.
Falls Anzahl = 1, dann: Falls oberes Bit des Ziels ungleich CF, dann OF = 1; anderenfalls OF = 0.
Falls Anzahl ungleich 1 ist, ist OF undefiniert.

Definierte Kennzeichenbits: CF,OF

Beschreibung: ROL rotiert die Bits des Bytes oder Wortes des Zieloperanden um die im Operanden „Anzahl" angegebene Zahl von Bits nach links. Falls der Wert von „Anzahl" gleich 1 ist, so kann er direkt angegeben werden. Wenn der Wert von Anzahl größer als 1 ist, muß er vor dem Gebrauch dieses Befehls ins CL-Register geladen werden.

Beispiele:

```
ROL BL,1
MOV CL, 2
RCL BL,CL
```

Vorher:	Nachher:
BL = 11001100, CF = 1	BL = 10011001, CF = 1 Anz. 1
BL = 10011001, CF = 1	BL = 00110011, CF = 1 Anz. 2

Codierung:

110100vw	mod 000 r/m

Falls v = 0, dann Anzahl = 1

ROR *Rotate Right* *Rechts rotieren*

Mnemonische Darstellung: ROR Ziel, Anzahl

Funktion: temp = Anzahl; Falls temp ungleich Null, dann: CF = unteres Bit des Ziels;
Ziel = Ziel / 2;
oberes Bit des Ziels = CF; temp = temp − 1; wiederhole Funktion bis temp = 0.
Falls Anzahl = 1, dann: Falls oberes Bit des Ziels ungleich zum nächstoberen Bit, dann OF = 1; sonst OF = 0.
Falls Anzahl ungleich 1 ist, ist OF undefiniert.

Beeinflußte Kennzeichenbits: CF,OF

Beschreibung: ROR rotiert die Bits des Bytes oder Wortes des Zieloperanden um die in „Anzahl" angegebene Zahl von Bits nach rechts. Falls der Wert von „Anzahl" gleich 1 ist, kann er direkt angegeben werden. Falls der Wert von „Anzahl" größer als 1 ist, muß er vor der Ausführung dieses Befehls ins CL-Register geladen werden.

Beispiele:

```
ROR BL,1
MOV CL,2
ROR BL,CL
```

Vorher:	Nachher:
BL = 11001100, CF = 0	BL = 01100110, CF = 0 Anz. 1
BL = 01100110, CF = 0	BL = 00110011, CF = 0 Anz. 2

Codierung:

110100vw	mod 001 r/m

Falls v = 0, dann Anzahl = 1

SAHF *Store Register AH Into Flags*
Register AH ins Kennzeichenregister übertragen

Mnemonische Darstellung: SAHF

Funktion: SF: ZF: X: AF: X: PF: X: CF:
d7 d6 d5 d4 d3 d2 d1 d0 AH übertragen in die Kennzeichenbits.

Definierte Kennzeichenbits: AF,CF,PF,SF,ZF

Beschreibung: SAHF überträgt die Bits 7,6,4,2 und 0 vom Register AH nach SF,ZF,AF,PF und CF, wobei die vorher in diesen Kennzeichenbits enthaltenen Werte überschrieben werden. OF,DF,IF und TF werden nicht beeinflußt.

Codierung:

10011110

SAL — *Shift Arithmetic Left* / *Arithmetisch links schieben*

SHL — *Shift Logical Left* / *Logisch links schieben*

Mnemonische Darstellung:

SAL Ziel, Anzahl
SHL Ziel, Anzahl

Funktion:

temp = Anzahl; Falls temp ungleich Null ist, dann: CF = oberes Bit des Ziels; Ziel = Ziel x 2;
unteres Bit des Ziels = 0; temp = temp − 1;
wiederhole Funktion bis temp = 0.
Falls Anzahl = 1, dann: Falls oberes Bit des Ziels ungleich zum CF benachbarten Bit ist, dann OF = 1; anderenfalls OF = 0.
Falls Anzahl ungleich 1 ist, ist OF undefiniert.

Definierte Kennzeichenbits:

CF,OF,PF,SF,ZF

Undefinierte Kennzeichenbits:

AF

Beschreibung:

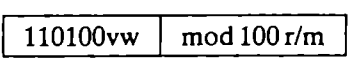
SHL und SAL führen dieselbe Operation durch und sind physikalisch derselbe Befehl. Das Zielbyte oder Wort wird um die Zahl der Bits, die im Operanden „Anzahl" angegeben ist, nach links geschoben. Nullen werden in die rechte Seite hineingeschoben. Falls das Vorzeichenbit seinen anfänglichen Wert behält, wird OF gelöscht. Falls der Wert von Anzahl gleich 1 ist, kann er direkt angegeben werden. Falls der Wert von Anzahl größer 1 ist, muß der Wert vor der Ausführung dieses Befehls in das CL-Register geladen werden.

Beispiele:

```
SAL AL, 1
MOV CL, 2
SAL AL, CL
```

Vorher:	Nachher:
AL = 11001100, CF = 0	AL = 10011000, CF = 1 Anz. 1
AL = 10011000, CF = 1	AL = 00110000, CF = 1 Anz. 2

Codierung:



Falls v = 0, dann Anzahl = 1

SAR

Shift Arithmetic Right
Arithmetisch rechts schieben

Mnemonische Darstellung: SAR Ziel, Anzahl

Funktion: temp = Anzahl; Falls temp ungleich Null ist, dann: CF = unteres Bit des Ziels; Ziel = Ziel/2;
oberes Bit des Ziels ist dasselbe wie das vorangegangene obere Bit (Vorzeichenbit); temp = temp − 1; wiederhole Funktion bis temp = 0.
Falls Anzahl = 1, dann: Falls das obere Bit des Ziels ungleich dem nächstfolgenden Bit ist, dann OF = 1; sonst OF = 0.

Definierte Kennzeichenbits: CF,OF,PF,SF,ZF

Undefinierte Kennzeichenbits: AF

Beschreibung: SAR verschiebt die Bits im Zieloperanden (Byte oder Wort) um die Zahl von Bits, die im Operanden Anzahl angegeben ist, nach rechts. Es werden Bits, die gleich dem ursprünglichen oberen (Vorzeichen-)Bit sind, in die linke Seite eingeschoben, wodurch das Vorzeichen des ursprünglichen Wertes erhalten bleibt. Beachten Sie, daß SAR nicht dasselbe Ergebnis wie ein „äquivalenter" IDIV-Befehl liefert, falls der Zieloperand negativ ist, und 1-Bits herausgeschoben werden.

Zum Beispiel ergibt −5 um ein Bit nach rechts geschoben −3, während die ganzzahlige Division −5/2 den Wert −2 liefert. Der Unterschied der Befehle liegt darin, daß IDIV alle Zahlen in Richtung 0 abschneidet, während SAR positive Zahlen in Richtung 0 abschneidet und negative Zahlen in Richtung negative Unendlichkeit.

Falls der Wert von „Anzahl" gleich 1 ist, kann er direkt angegeben werden. Falls der Wert von „Anzahl" größer als 1 ist, muß er vor der Ausführung dieses Befehls ins CL-Register geladen werden.

Beispiele:

```
SAR BL, 1
MOV CL, 2
SAR AL, CL
```

Vorher:	Nachher:
BL = 11001100, CF = 0	BL = 11100110, CF = 0 Anz. 1
BL = 11100110, CF = 0	BL = 11110011, CF = 0 Anz. 2

Codierung:

110100vw	mod 111 r/m

Falls v = 0, dann Anzahl = 1

SBB

Subract With Borrow
Mit geborgtem Übertrag subtrahieren

Mnemonische Darstellung: SBB Ziel, Quelle

Funktion: Falls CF = 1, dann Ziel = Ziel − Quelle − 1
Falls CF ungleich 1, dann Ziel = Ziel − Quelle

Definierte Kennzeichenbits: AF,CF,OF,PF,SF,ZF

Beschreibung: SBB subtrahiert die Quelle vom Ziel, subtrahiert dann 1, falls CF gesetzt ist, und bringt das Ergebnis in den Zieloperanden. Beide Operanden können Bytes oder Wörter sein. Beide Operanden können binäre Zahlen mit oder ohne Vorzeichen sein.

Codierung: Speicher- oder Registeroperand und Registeroperand

000110dw	mod reg r/m

Unmittelbarer Operand von Speicher- oder Registeroperand

100000sw	mod 011 r/m	Daten	Daten, falls s:w = 01

Unmittelbarer Operand von Akkumulator

0001110w	Daten	Daten, falls w = 1

SCAS *Scan (Byte Or Word) String*
(Byte- oder Wort-) Zeichenkette untersuchen

Mnemonische Darstellung: SCAS Zielzeichenkette

Funktion: Falls die Zielzeichenkette aus Bytes besteht, dann werden die Daten, die durch das DI-Register adressiert werden, von AL subtrahiert. Falls DF = 0, dann DI = DI + 1; falls DI = 1, dann DI = DI − 1.
Falls die Zielzeichenkette aus Worten besteht, dann werden die Daten, die durch das DI-Register adressiert werden, von AX subtrahiert. Falls DF = 0, dann DI = DI + 2; falls DF = 1, dann DI = DI − 2.

Definierte Kennzeichenbits: AF,CF,OF,PF,SF,ZF

Beschreibung: SCAS subtrahiert das durch DI adressierte Zeichenkettenelement (Byte oder Wort) vom Inhalt von AL (Zeichenkette aus Bytes) bzw. AX (Zeichenkette aus Wörtern) und aktualisiert die Kennzeichenbits, aber verändert weder die Zielzeichenkette noch den Akkumulator. SCAS aktualisiert ebenfalls DI als Zeiger auf das nächste Zeichenkettenelement und AF,CF, OF,PF,SF und ZF, um die Beziehung zwischen dem Wert in AL/AX und dem „abgetasteten (scanned)" Zeichenkettenelement widerzuspiegeln. Dem SCAS-Befehl kann REPE, REPZ, REPNE oder REPNZ vorangestellt werden.

Codierung:

1010111w

SHR *Shift Logical Right* *Logisch rechts schieben*

Mnemonische Darstellung: SHR Ziel, Anzahl

Funktion: temp = Anzahl; Falls temp ungleich Null ist, dann: CF = unteres Bit des Ziels; Ziel = Ziel/2;
oberes Bit des Ziels = 0; temp = temp − 1;
wiederhole Funktion bis temp = 0.
Falls Anzahl = 1, dann: Falls das obere Bit des Ziels ungleich dem nächstfolgenden Bit ist, dann OF = 1; sonst OF = 0.
Falls Anzahl ungleich 1 ist, ist OF undefiniert.

Definierte Kennzeichenbits: CF, OF, PF, SF, ZF

Undefinierte Kennzeichenbits: AF

Beschreibung: SHR verschiebt die Bits im Zieloperanden (Byte oder Wort) um die in dem Operanden „Anzahl" angegebene Zahl nach rechts. Nullen werden in die linke Seite eingeschoben. Falls das Vorzeichenbit seinen ursprünglichen Wert behält, wird OF gelöscht.

Falls der Wert von „Anzahl" = 1 ist, kann er direkt angegeben werden. Falls der Wert größer als 1 ist, muß er vor der Ausführung dieses Befehls ins CL-Register geladen werden.

Beispiele:

```
SHR BL,1
MOV CL,2
SHR BL,CL
```

Vorher:	Nachher:
BL = 00110011, CF = 0	BL = 00011001, CF = 1 Anz. 1
BL = 00011001, CF = 1	BL = 00001100, CF = 1 Anz. 2

Codierung:

110100vw	mod 101 r/m

Falls v = 0, dann Anzahl = 1.

STC

Set Carry
Übertragskennzeichen setzen

Mnemonische Darstellung: STC

Funktion: CF = 1

Definierte Kennzeichenbits: CF

Beschreibung: STC setzt CF auf 1 und beeinflußt keine anderen Kennzeichenbits.

Codierung:

11111001

STD

Set Direction Flag
Richtungskennzeichen setzen

Mnemonische Darstellung: STD

Funktion: DF = 1

Definierte Kennzeichenbits: DF

Beschreibung: STD setzt DF auf 1, wodurch bei Zeichenkettenbefehlen das SI- und/oder das DI-Register automatisch erniedrigt wird. STD beeinflußt keine anderen Kennzeichenbits. Falls das DF-Bit nicht gesetzt ist, wird bei Zeichenkettenbefehlen automatisch inkrementiert.

Codierung:

11111101

STI

Set Interrupt Enable Flag
Unterbrechungsfreigabekennzeichen setzen

Mnemonische Darstellung: STI

Funktion: IF = 1

Definierte Kennzeichenbits: IF

Beschreibung: STI setzt IF auf 1 und gestattet der CPU maskierbare Unterbrechungsanforderungen auf der INTR-Eingabeleitung zu erkennen. Eine bereits anliegende Unterbrechung wird nicht erkannt, bevor der auf STI folgende Befehl ausgeführt wurde. STI beeinflußt keine anderen Kennzeichenbits.

Codierung:

11111011

STOS *Store (Byte or Word) String*
(Byte- oder Wort-) Zeichenketten speichern

Mnemonische Darstellung: STOS Zielzeichenkette

Funktion: Falls ein Byte vorliegt, wird der Wert in AL unter der Adresse, auf die DI zeigt, gespeichert.
Falls FD = 0, dann DI = DI + 1. Falls DF = 1, dann DI = DI − 1.

Falls ein Wort vorliegt, wird der Wert in AX unter der Adresse, auf die DI zeigt, gespeichert.
Falls FD = 0, dann DI = DI + 2. Falls DF = 1, dann DI = DI − 2.

Beeinflußte Kennzeichenbits: keine

Beschreibung: STOS überträgt ein Byte oder Wort vom Register AL oder AX zu dem durch DI adressierten Zeichenkettenelement und aktualisiert DI als Zeiger auf die nächste Position in der Zeichenkette.

Beispiele:
```
STOS BYTE_ZIEL
STOS WORT_ZIEL
REP STOS BYTE_ZIEL1
```

Codierung:

1010101w

SUB *Subtract* *Subtrahieren*

Mnemonische Darstellung: SUB Ziel, Quelle

Funktion: Ziel = Ziel − Quelle

Definierte Kennzeichenbits: AF,CF,OF,PF,SF,ZF

Beschreibung: Der Quelloperand wird vom Zieloperanden subtrahiert, und das Resultat überschreibt den Zieloperanden. Die Operanden können Bytes oder Wörter sein. Beide Operanden können binäre Zahlen mit oder ohne Vorzeichen sein. Beide Operanden können binäre Zahlen mit oder ohne Vorzeichen sein.

Beispiele:
SUB BX,AX
SUB BL, 10
SUB SI, 4790

Codierung: Speicher- oder Registeroperand und Registeroperand

001010dw	mod reg r/m

Unmittelbarer Operand von Speicher- oder Registeroperand

100000sw	mod 101 r/m	Daten	Daten, falls s:w = 01

Unmittelbarer Operand von Akkumulator

0010110w	Daten	Daten, falls w = 1

TEST *Test (Logical Compare)*
Testen (logisch vergleichen)

Mnemonische Darstellung: TEST Ziel, Quelle

Funktion: Das Ziel wird mit der Quelle durch das logische „Und" verknüpft. Die Kennzeichenbits werden aktualisiert. CF = 0; OF = 0.

Definierte Kennzeichenbits: CF,OF,PF,SF,ZF

Undefinierte Kennzeichenbits: AF

Beschreibung: TEST führt das logische „Und" der beiden Operanden (Bytes oder Wörter) durch und aktualisiert die Kennzeichenbits, bringt das Ergebnis aber nicht zurück (d.h. keiner der Operanden wird verändert). Falls einem Testbefehl ein JNZ-Befehl (jump if not zero, Sprung, falls nicht null) folgt, wird der Sprung durchgeführt, wenn es irgendwelche korrespondierenden 1-Bits in beiden Operanden gibt.

Beispiele:
TEST AL, 3Fh
TEST Bx, 0557h

Codierung: Speicher- und Registeroperand mit Registeroperand

1000010w	mod reg r/m

Unmittelbarer Operand mit Speicher- oder Registeroperand

1111011w	mod 000 r/m	Daten	Daten, falls w = 1

Unmittelbarer Operand mit Akkumulator

1010100w	Daten	Daten, falls w = 1

WAIT *Wait* *Warten*

Mnemonische Darstellung: WAIT

Funktion: keine

Beeinflußte Kennzeichenbits: keine

Beschreibung: WAIT veranlaßt die CPU so lange in den Wartezustand zu gehen, wie ihr TEST-Eingang nicht aktiv ist.

Codierung: 10011011

XCHG *Exchange* *Austauschen*

Mnemonische Darstellung: XCHG Ziel, Quelle

Funktion: temp = Ziel; Ziel = Quelle; Quelle = temp

Beeinflußte Kennzeichenbits: keine

Beschreibung: XCHG vertauscht die Inhalte von Quell- und Zieloperanden (Bytes oder Wörter).

Beispiele:
XCHG BX,AX
XCHG Marke,CX

Codierung: Speicher- oder Registeroperand mit Registeroperand

1000011w	mod reg r/m

Registeroperand mit Akkumulator

10010 reg

XLAT *Translate Übersetzen*

Mnemonische Darstellung: XLAT Tafel_Name

Funktion: AL = Wert an der Adresse, auf die BX + AL zeigt.

Beeinflußte Kennzeichenbits: keine

Beschreibung: XLAT ersetzt ein Byte im AL-Register durch ein Byte aus einer 256 byte großen, benutzerdefinierten Übersetzungstabelle. Es wird angenommen, daß das Register BX auf den Anfang der Tabelle zeigt. Das Byte in AL wird als Index in der Tabelle benutzt und wird ersetzt durch das Byte des Offsets in der Tabelle, der dem binären Wert von AL entspricht. Das erste Byte der Tabelle hat ein Offset von Null.

Wenn zum Beispiel AL den Wert 5h enthält, und das sechste Element der Übersetzungstabelle 33h enthält, so wird AL nach der Ausführung von XLAT den Wert 33h enthalten.

Beispiel:

```
MOV BX, OFFSET_Tabellen_Wert
XLAT Tabelle_1
```

Codierung:

11010111

XOR *Exclusive Or*
Exklusives Oder bilden

Mnemonische Darstellung: XOR Ziel, Quelle

Funktion: Ziel = Ziel ∀ Quelle; CF = 0; OF = 0.

Definierte Kennzeichenbits: CF,OF,PF,SF,ZF

Undefinierte Kennzeichenbits: AF

Beschreibung: XOR führt das logische „Exclusiv-Oder" der beiden Operanden durch und bringt das Ergebnis in den Zieloperanden. Ein Bit in dem Ergebnis wird gesetzt, falls die korrespondierenden Bits der ursprünglichen Operanden entgegengesetzte Werte haben.

Beispiel: XOR AX,BX

Vorher:	Nachher
AX = 5857H	AX = 00FFH
BX = 58A8H	BX = 58A8H

Codierung: Speicher- oder Registeroperand mit Registeroperand

001100dw	mod reg r/m

Unmittelbarer Operand zu Speicher- oder Registeroperand

1000000w	mod 110 r/m	Daten	Daten, falls w = 1

Unmittelbarer Operand zu Akkumulator

0011010w	Daten	Daten, falls w = 1

Kapitel 5

Grundlegende Programmiertechniken

In diesem Kapitel untersuchen wir einige grundlegende Techniken zur Programmierung des 8086/8088. Wir schreiben einfache arithmetische Programme unter Verwendung vieler in Kapitel 4 aufgeführter Befehle. Außerdem behandeln wir Unterprogramme und Rücksprünge aus Unterprogrammen. Wir lernen, daß es aufgrund der Breite des 8086/8088-Adreßbusses mehrere verschiedene Möglichkeiten gibt, ein Unterprogramm aufzurufen.

ARITHMETISCHE PROGRAMME

Die arithmetischen Programme in diesem Kapitel zeigen Ihnen, wie Additionen, Subtraktionen, Multiplikationen und Divisionen mit dem 8086/8088 ausgeführt werden. Diese Programme lehren Sie ganzzahlige arithmetische Operationen mit positiven binären Zahlen und negativen Zahlen, die im Zweier-Komplement dargestellt sind, auszuführen. Wir beginnen mit einem Beispiel einer 8-Bit-Addition. (Beachten Sie jedoch, daß der 8086/8088 auch in der Lage ist, arithmetische Operationen mit 16-Bit-Zahlen auszuführen.)

8-Bit-Addition

Das folgende Additionsprogramm addiert zwei 8-Bit-Operanden, OP1 und OP2, die an den Speicheradressen ADR1 und ADR2 gespeichert sind. Die Summe, die in der Speicherzelle ADR3 gespeichert ist, heißt RES. Abbildung 5.1 zeigt in einem Blockdiagramm, wie die Operanden und das Ergebnis gespeichert werden.

Hier ist das Programm:

```
Befehl              Kommentar
MOV AL, ADR1        ;BRINGE OP1 NACH AL
ADD AL, ADR2        ;ADDIERE AL UND OP2 @ ADR2
MOV ADR3, AL        ;SPEICHERE DIE SUMME @ ADR3
```

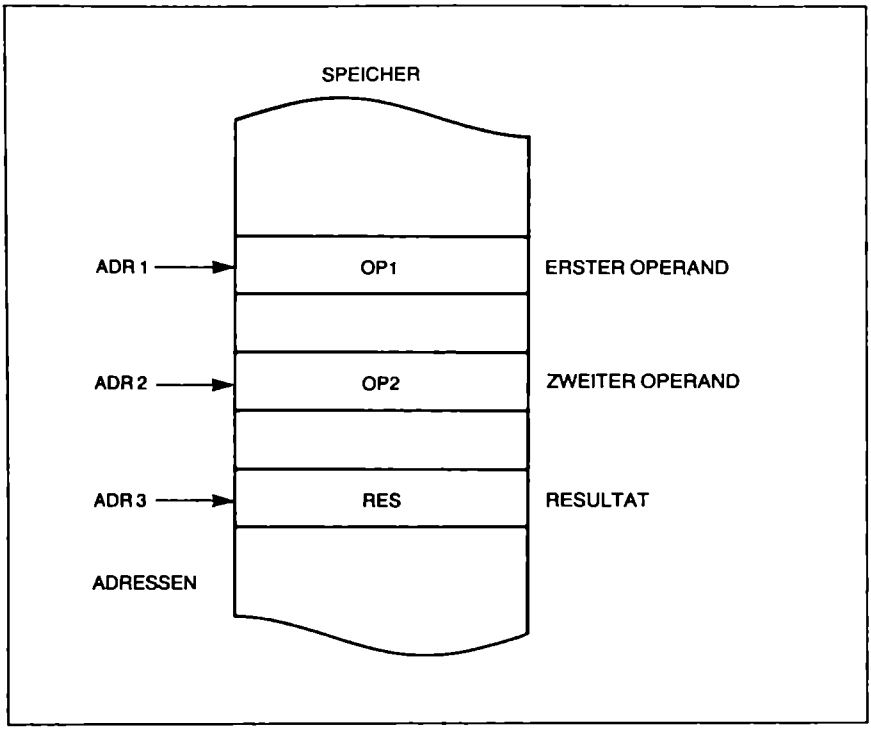


Abb. 5.1: Dieses Diagramm zeigt, wie die Operanden und das Ergebnis im Speicher gespeichert werden müssen. OP1 wird zu OP2 addiert und an der Speicherposition RES gespeichert

Jede Zeile des Programms (das hier in symbolischer Form ausgedrückt ist) beinhaltet einen *Befehl*. Jeder Befehl wird von dem Computer in den entsprechenden Objektcode übersetzt, den die CPU ausführen kann. Später in dieser Diskussion werden wir den Objektcode für dieses Programm untersuchen. Lassen Sie uns nun einen Blick auf das Programm werfen.

Die erste Zeile besagt, daß die Daten der Speicherzelle ADR1 in das Register AL (8 Bit) geladen werden. (Wir hätten jedes der allgemeinen 8-Bit-Register in der 8086/8088 CPU anstelle des AL-Registers wählen können.)

ADR1 ist eine symbolische Darstellung der tatsächlichen Adresse im Systemspeicher. Diese Adresse ist anderswo im Programm definiert. Wenn die CPU die Daten aus dem Speicher liest, werden der 16-Bit-Wert von ADR1 und das entsprechende Segmentregister summiert, um die 20-Bit-Systemadresse zu erzeugen.

Lassen Sie uns auf das Programm zurückkommen und den KOMMENTAR-Teil, den rechten Teil jeder Befehlszeile, untersuchen. Kommentare sind für den Programmierer sehr nützlich, um zu verstehen und sich genau zu erinnern, was jede Zeile des Programms bewirkt. Die Kommentare werden bei der Erzeugung des Objektcodes ignoriert. Abb. 5.2 zeigt die Ergebnisse nach der Ausführung des ersten Befehls.

Der zweite Befehl

```
ADD AL, ADR2
```

besagt, daß die Daten der Speicheradresse ADR2 zum Inhalt des 8-Bit-Registers AL addiert werden. Die Adresse ADR2 enthält den zweiten Operanden, OP2. Bei der Ausführung des zweiten Befehls wird OP2 aus dem Speicher gelesen und zu OP1 addiert. Das Ergebnis wird im AL-Register gespeichert.

Die Summe von OP1 und OP2 ist nun also im AL-Register enthalten. Um das Programm abzuschließen, müssen wir den Inhalt von AL in die Speicherzelle ADR3 bringen. Das wird durch die dritte Zeile des Programms erreicht:

```
MOV ADR3, AL
```

An diesem einfachen Programm können wir erkennen, daß es einige wichtige Punkte bei der Arbeitsweise des 8086/8088 gibt, die wir behandeln sollten. Ein solcher Punkt ist, daß die Daten byteweise gespeichert werden. So ist in diesem Beispiel, nachdem das Ergebnis in den Speicher nach ADR3 geschrieben wurde, nur das Byte ADR3 verändert. Das ist trotz des 16-Bit-Datenbusses des 8086 richtig. Die CPU überschreibt nur ein einziges Byte im Systemspeicher, weil in dem Befehl ein 8-Bit-Register verwendet wird.

Ein anderer interessanter Punkt in dem Programm ist, daß Daten aus dem Speicher in ein internes Register gebracht werden. Die Speicherzelle, aus der die Daten gelesen werden, heißt *Quelloperand*. Dieser Operand wird während der Ausführung des Programms nicht verändert. Wenn wir die Speicherzellen ADR1 und ADR2 nach Beendigung des Programms untersuchen, stellen wir fest, daß sie dieselben Daten wie zu Beginn des Programms enthalten.

Lassen Sie uns nun fortfahren und den Adressen der Speicherzellen ADR1, ADR2 und ADR3 tatsächlich numerische Werte zuweisen. Das können wir durch die Verwendung von „Pseudobefehlen" tun. Pseudobefehle sind Befehle, die nicht vom Mikroprozessor, sondern von dem Computerprogramm, das den Objektcode erzeugt, verwendet werden. Der Pseudobefehl, den wir hier benutzen, ist der Equate-Befehl – geschrieben als EQU. Der Name ist aus seiner Funktion abgeleitet (equate –

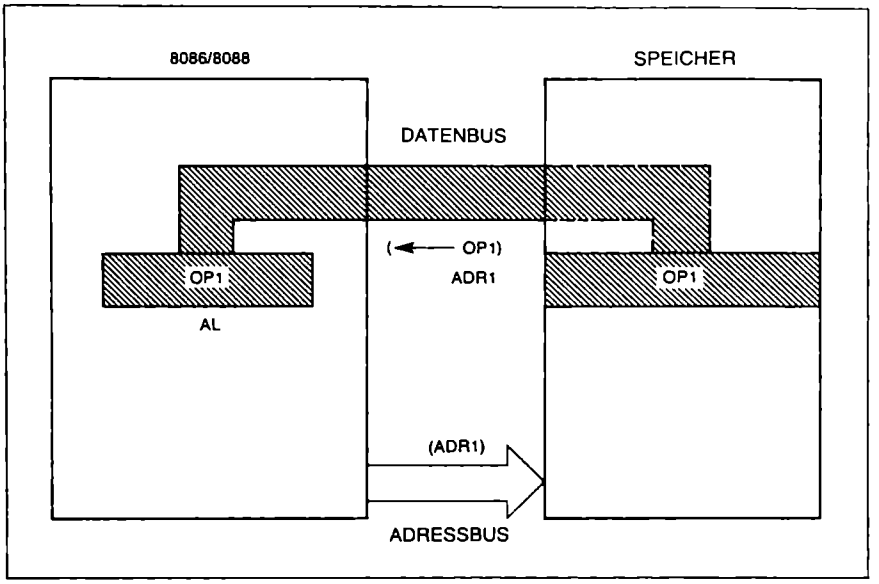


Abb. 5.2: Nachdem der Befehl MOV AL, OP1 ausgeführt ist, enthalten die unteren 8 Bits des AX-Registers den Wert, der an der Speicherposition OP1 gespeichert ist

gleichsetzen). Dieser Befehl setzt die Argumente auf beiden Seiten des Pseudobefehls EQU gleich. Ein vollständiges Programm für diese 8-Bit-Addition zeigt Abbildung 5.3. In diesem Programm sind ADR1, ADR2 und ADR3 gleich 300H, 320H und 345H.

```
                    1;
                    2;EIN PROGRAMM ZUR ADDITION ZWEIER 8-BIT-ZAHLEN
                    3;
                    4 ORG 0200H
                    5;
                    6 ADR1          EQU 0300H
                    7 ADR2          EQU 0320H
                    8 ADR3          EQU 0345H
                    9;
0200 0A0003         10 MOV AL,ADR1      ;LADE AL MIT DATEN @ ADR1
0203 02062003       11 ADD AL,ADR2      ;ADDIERE AL MIT DATEN @ ADR2
0207 A24503         12 MOV ADR3,AL      ;SPEICHERE AL @ ADR3
                    13;
```

Abb. 5.3: Dies ist ein assembliertes 8086/8088-Programm zur Addition zweier 8-Bit-Zahlen.

16-Bit-Addition

Diese 8-Bit-Additionsaufgabe wollen wir nun zu einer 16-Bit-Additionsaufgabe ausweiten. Ein 16-Bit-Problem kann auf genau die gleiche Art wie ein 8-Bit-Problem gelöst werden. Einen Unterschied gibt es nur bei den verwendeten Registern und der Anzahl der benötigten Speicherzellen.

Erinnern Sie sich, daß beim 8086/8088 ein Byte eine einzige Speicherzelle ist. So erfordert ein Wort zwei Speicherzellen. Abb. 5.4 zeigt,

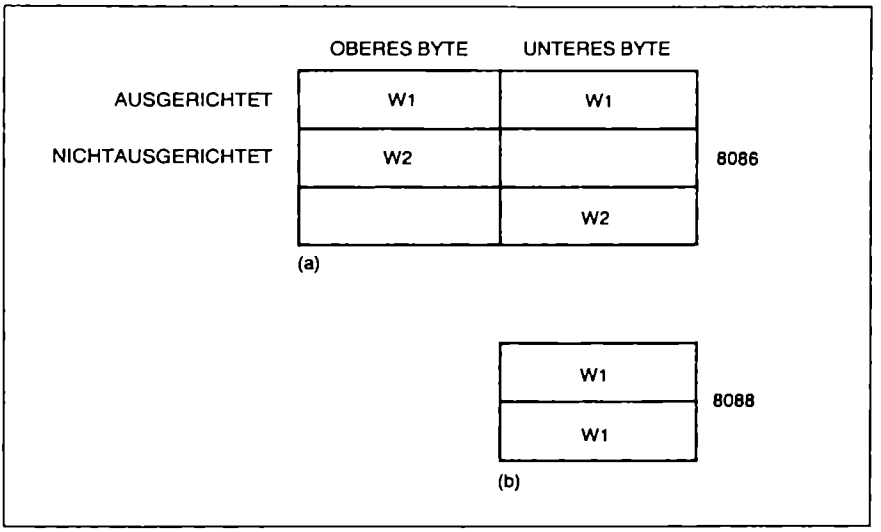


Abb. 5.4a: 16-Bit-Daten in einem 8086-System können ausgerichtet oder nichtausgerichtet sein

Abb. 5.4b: 16-Bit-Daten in einem 8088-System benötigen zwei aufeinanderfolgende Bytes des Speichers

wie die Daten für 16-Bit-Operationen im 8086/8088 Systemspeicher organisiert sind. Ein Programm zur Addition zweier 16-Bit-Zahlen ist in Abb. 5.5 dargestellt.

16-Bit-Subtraktion

Die Subtraktion zweier 16-Bit-Zahlen mit dem 8086/8088 ist ähnlich zu dem 16-Bit-Additionsproblem, das wir eben gelöst haben. Unter Benutzung derselben Zahlen subtrahieren wir nun OP2 von OP1 und speichern das Ergebnis in ADR3. Die Daten sind wie in Abb. 5.6 gezeigt im Speicher organisiert. Ein Programm zur Ausführung der 16-Bit-Subtraktion ist in Abb. 5.7 dargestellt.

```
                    1;
                    2;EIN PROGRAMM ZUR ADDITION ZWEIER 16-BIT-ZAHLEN
                    3;
                    4 ORG 0250H
                    5;
                    6 ADR1          EQU 0300H
                    7 ADR2          EQU 0320H
                    8 ADR3          EQU 0345H
                    9;
0250 A10003         10 MOV AX,ADR1      ;LADE AX MIT DATEN @ ADR1
0253 03062003       11 ADD AX,ADR2      ;ADDIERE AX MIT DATEN @ ADR2
0257 A34503         12 MOV ADR3,AX      ;SPEICHERE AX @ ADR3
                    13;
```

Abb. 5.5: Dieses assemblierte 8086/8088-Programm zur Addition zweier 16-Bit-Zahlen ist ähnlich dem 8-Bit-Programm in Abb. 5.3, mit der Ausnahme, daß das AX-Register anstelle des AL-Registers benutzt wird

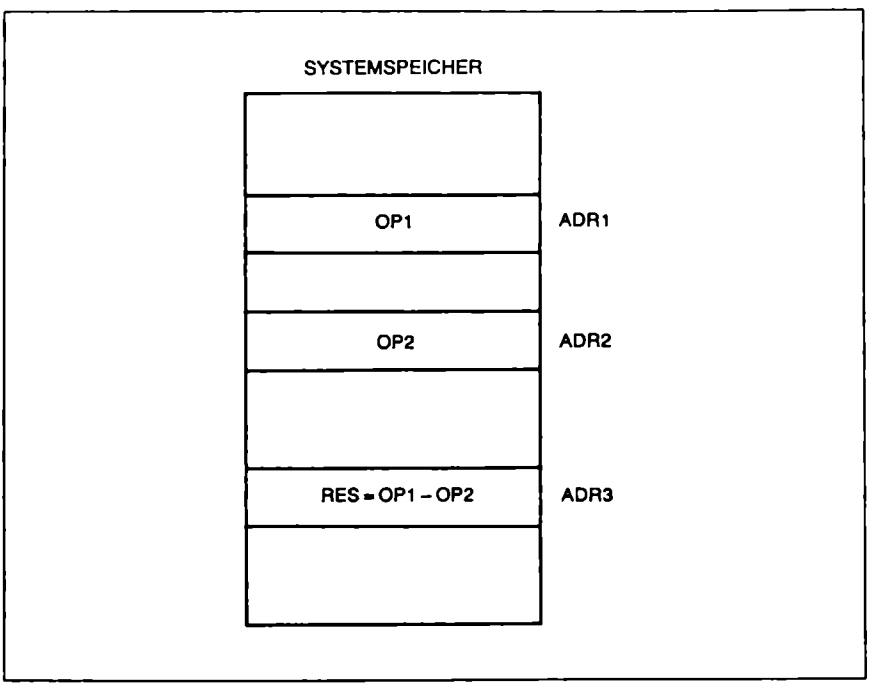


Abb. 5.6: Dieses Speicherdiagramm zeigt, wie die Operanden (OP1 und OP2) und das Resultat in einem Subtraktionsbeispiel auftreten

```
                    1 ,
                    2 ; EIN PROGRAMM ZUR SUBTRAKTION ZWEIER 16-BIT-ZAHLEN
                    3 ;
                    4 ORG 0250H
                    5 ;
                    6 ADR1          EQU 0300H
                    7 ADR2          EQU 0320H
                    8 ADR3          EQU 0345H
                    9 ;
0250 A10003         10 MOV AX,ADR1      ;LADE AX MIT DATEN @ ADR1
0253 2B062003       11 SUB AX,ADR2      ;SUBTRAHIERE VON AX MIT DATEN @ ADR2
0257 A34503         12 MOV ADR3,AX      ;SPEICHERE AX @ ADR3
                    13 ;
```

Abb. 5.7: Dies ist ein assembliertes 8086/8088-Programm zur Durchführung einer 16-Bit-Subtraktion

BCD-ARITHMETIK

In Kapitel 1 haben wir das Konzept der BCD-Arithmetik behandelt. Rufen Sie sich ins Gedächtnis zurück, daß BCD in Anwendungen benutzt wird, in denen es unbedingt erforderlich ist, daß jede signifikante Stelle des Ergebnisses erhalten bleibt.

8-Bit-BCD-Addition

In BCDNotation wird ein 4-Bit-Nibble benutzt, um eine Dezimalstelle zu speichern (0–9). Daraus folgt, daß jedes 8-Bit-Byte zwei BCDZiffern speichern kann. (Dies heißt *gepacktes BCD*.) Nun wollen wir sehen, wie BCD arbeitet. Lassen Sie uns zwei Bytes addieren, wobei jedes zwei BCD-Ziffern enthält. Ein Speicherdiagramm dieses Beispiels ist in Abb. 5.8 dargestellt.

Bevor wir irgendwelche Programme zur Ausführung von BCD-Additionen schreiben, wollen wir zuerst mit einigen Zahlenbeispielen arbeiten. Das wird uns helfen, Probleme, die bei dieser Art der Addition möglicherweise entstehen, zu formulieren. Zuerst wollen wir 01 und 02 addieren:

01 *wird dargestellt durch*:	0000 0001
02 *wird dargestellt durch*:	0000 0010
Das Ergebnis ist:	0000 0011

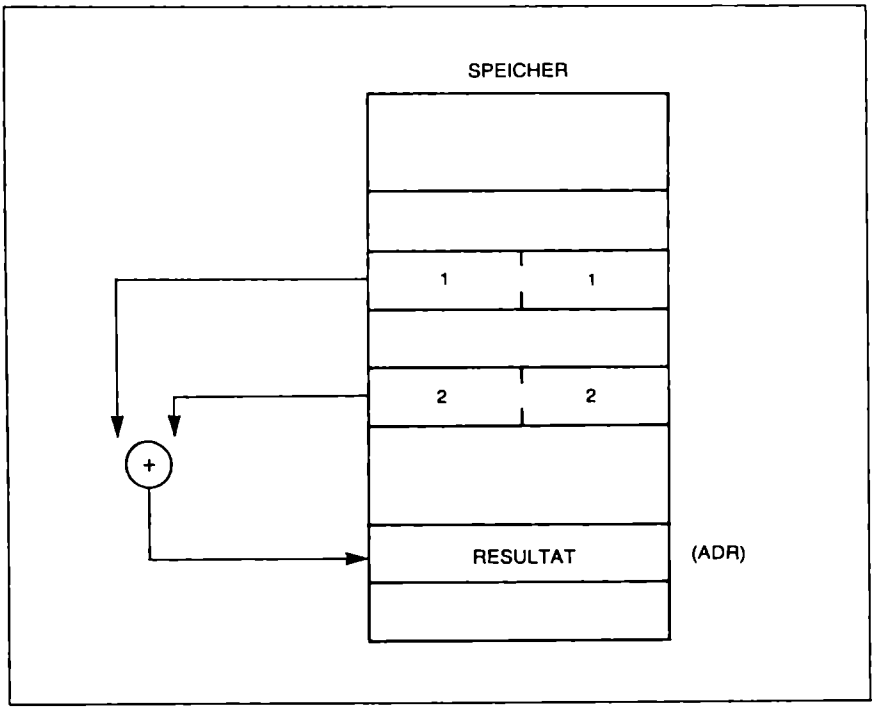


Abb. 5.8: Dieses Speicherdiagramm zeigt die Speicherung zweier gepackter BCD-Bytes vor und nach der Addition

Dieses Ergebnis ist die BCD-Darstellung für 03. (Wenn Sie sich des BCD-Äquivalents nicht sicher sind, schlagen Sie in der Umrechnungstabelle am Ende dieses Buches nach.) In diesem Beispiel ging alles ganz einfach; deshalb lassen Sie uns ein anderes ausprobieren. Wir wollen 8 und 3 addieren:

08 *wird dargestellt durch*: 0000 1000
03 *wird dargestellt durch*: 0000 0011

Wenn Sie das Ergebnis 0000 1011 erhalten haben, haben Sie die binäre Summe von 8 und 3 berechnet. Diese Zahl ist in der Tat die Dezimalzahl 11 in binärer Darstellung. Leider ist 1011 in BCD ein unzulässiger Code. Die BCD-Darstellung von 11 ist 0001 0001. Dieser Unterschied liegt in der Tatsache begründet, daß die BCD-Darstellung nur die ersten 10 Kombinationen von vier Stellen benutzt, um die dezimalen Symbole 0-9 zu verschlüsseln. Infolgedessen sind die verbleibenden sechs möglichen Kombinationen in der BCD-Notation unbenutzt. Die Zahl 1011 ist eine dieser unbenutzten Kombinationen. Mit anderen Worten: Immer, wenn

die Summe von zwei BCD-Ziffern größer als 9 ist, müssen sie 6 zu dem Ergebnis addieren, um die 6 unbenutzten Codes zu überspringen.

Das folgende Beispiel zeigt, wie das vorangegangene Beispiel zu korrigieren ist. Durch einfaches Addieren von 6 zu dem binären Äquivalent von 11 (oder der Summe, die wir gerade durch Addition von 8 und 3 erhalten haben), erhalten wir das richtige Ergebnis:

```
    1011     unzulässiges BCD-Ergebnis
    0110     + 6
---------
0001 0001
```

Dieses Ergebnis ist die Dezimalzahl 11 in BCD-Darstellung. Nun haben wir die richtige Antwort auf das BCD-Additionsproblem von 8+3.

Dieses Beispiel veranschaulicht eine der grundlegenden Schwierigkeiten des BCD-Modus: Sie müssen die sechs fehlenden Codes ausgleichen. Es gibt jedoch einen speziellen Additionsanpassungsbefehl (DAA), der das Ergebnis binärer Additionen automatisch anpaßt. (Dieser Befehl addiert 6, wenn das Ergebnis größer als 9 ist.)

Das gleiche Beispiel wollen wir nun benutzen, um einen anderen Punkt der BCD-Arithmetik zu veranschaulichen. Wenn wir 6 zu dem Ergebnis 11 addieren, wird ein Übertrag vom unteren Nibble zum oberen Nibble erzeugt. Dieser interne Übertrag muß berücksichtigt und zu der zweiten BCD-Ziffer addiert werden. Obwohl der Additionsbefehl dieses Problem automatisch löst, ist es oft nützlich, wenn man diesen Übertrag von Bit 3 nach Bit 4 direkt erkennen kann. Dies wird durch Überprüfung des AF-Kennzeichens ermöglicht. (Dieser Übertrag wird manchmal als Halbübertrag oder Hilfsübertrag bezeichnet.)

Als ein Beispiel hier die Programmbefehle zur Addition zweier BCDZahlen, 11 und 22:

```
MOV AL, 11H     ;11 IN DAS AL-REGISTER BRINGEN
ADD AL, 22H     ;ADDIERE 11 UND 22 UND SPEICHERE
                ;DAS ERGEBNIS IN AL
DAA             ;PASSE DAS ERGEBNIS AN
MOV ADR3, AL    ;SPEICHERE DAS ERGEBNIS
```

(Das H hinter den Zahlen 11 und 22 gibt an, daß diese Zahlen hexadezimal anstatt dezimal zu interpretieren sind.) Das AL-Register wird verwendet, weil der DAA-Befehl erwartet, daß der Operand in diesem Register steht.

Dieses Programm ist ähnlich zu dem, das wir für die binäre 8-Bit-Addition angegeben haben. Es verwendet jedoch einen neuen Befehl: DAA (Dezimale Anpassung für die Addition). (Für weitere Informationen über den

DAA-Befehl schlagen Sie bitte in Kapitel 4 nach). Lassen Sie uns nun genau anschauen, was der DAA-Befehl in diesem Programm tut.

Das erste Ereignis, das eintritt, ist daß die Zahlen 11 und 22 addiert werden. Dies erscheint in dem AL-Register so:

```
  0001 0001  (11)
+ 0010 0010  (22)
= 0011 0011  (33)
```

Da die Zahl in den unteren 4 Bits des Ergebnisses nicht größer als 9 ist und es keinen Übertrag von Bit 3 nach Bit 4 gab, hat der DAA-Befehl nichts an den unteren 4 Bits des Ergebnisses verändert. (AF ist gleich 0.) Weiter gibt es keinen Anlaß, die oberen 4 Bits des Ergebnisses zu ändern. Deshalb hat der DAA-Befehl das Ergebnis in diesem Beispiel überhaupt nicht verändert.

Lassen Sie uns nun ein anderes Beispiel nehmen. Wir wollen die Zahlen 22 und 39 addieren:

```
  0010 0010  (22)
+ 0011 1001  (39)
= 0101 1011  (5?)
```

1011 ist ein unzulässiger BCD-Code, da es größer als 9 ist. Das Ergebnis muß korrigiert werden. Wir können den DAA-Befehl benutzen, der 6 zu den unteren 4 Bits addiert und einen Übertrag in die nächsten 4 Bits erzeugt. Dies führt zu folgendem Ergebnis:

```
  0101 1011  (5?)
+ 0000 0110  (06)
= 0110 0001  (61)
```

Dieses ist das richtige Ergebnis für die Addition von 22 und 39. In diesem Fall wurde der DAA-Befehl benötigt, um die richtige Lösung zu erzeugen.

BCD-Subtraktion

Die Ausführung einer BCD-Subtraktion mit dem 8086/8088 ist ebenso einfach wie die Ausführung einer BCD-Addition. Dies liegt an der Tatsache, daß die CPU einen speziellen DAS-Befehl (dezimale Anpassung für die Subtraktion) hat. (Für weitere Informationen über diesen Befehl schlagen Sie bitte in Kapitel 4 nach.) Hier ist ein Programm zur Subtraktion zweier gepackter BCD-Zahlen:

```
MOV AL, 42H     ;LADE DIE HEXADEZIMALZAHL 42
                ;IN DAS REGISTER AL
SUB AL, 23H     ;SUBTRAHIERE 23H VON 42H
DAS             ;PASSE DAS ERGEBNIS AN
MOV ADR3, AL    ;SPEICHERE DAS ERGEBNIS
```

Bisher haben wir Beispiele zur Addition und Subtraktion angegeben unter Verwendung verschiedener 8086/8088 Befehle. Lassen Sie uns nun die arithmetischen Operationen Multiplikation und Division untersuchen.

Multiplikation

Lassen Sie uns unsere Untersuchung der Multiplikation mit einer einfachen dezimalen Multiplikationsaufgabe beginnen. Wir wollen 12 mit 23 multiplizieren:

```
  12   Multiplikand
× 23   Multiplikator
  36   1. Teilprodukt
+ 24   2. Teilprodukt
 276   Endergebnis
```

Die Multiplikation wird ausgeführt, indem zuerst die äußerste rechte Ziffer des Multiplikators mit dem Multiplikanden multipliziert wird, d.h. 3 x 12 (das Teilprodukt ist 36), anschließend die nächste Ziffer des Multiplikators (2) mit 12 multipliziert wird und indem schließlich diese beiden Ergebnisse zur Berechnung des Endergebnisses addiert werden.

Es gibt jedoch eine Operation mehr: Entweder muß die 24 um eine Stelle nach links verschoben werden oder, was gleichbedeutend ist, das Teilprodukt (36) muß um eine Stelle nach rechts verschoben werden. Dann werden die beiden Zahlen addiert. Die Summe ist 276.

Lassen Sie uns nun die binäre Multiplikation untersuchen. Sie wird in derselben Weise wie die dezimale Multiplikation durchgeführt. Wir wollen 5 x 3 berechnen:

```
(5)    101   Multiplikand
(3)  × 011   Multiplikator
       101   1. Teilprodukt
      101    2. Teilprodukt
    + 000    3. Teilprodukt
(15) 01111   Endergebnis
```

An diesem Beispiel können wir sehen, daß die binäre Multiplikation der dezimalen Multiplikation sehr ähnlich ist. Wir haben Glück, daß der 8086/8088 mit einem Befehl ausgerüstet ist, der diese Multiplikation für uns

ausführt. Frühe Mikroprozessoren, ebenso wie viele 8-Bit-Mikroprozessoren, haben keinen Multiplikationsbefehl. Deshalb wird die Multiplikation durch ein kleines Programm ausgeführt. Tatsächlich sind das Addieren und Verschieben die einzigen arithmetischen Operationen, die diese Mikroprozessoren ausführen können.

Lassen Sie uns nun untersuchen, wie die Multiplikation auf dem 8086/8088 durchgeführt wird.

Multiplikation von 16-Bit-Zahlen

Mit dem 8086/8088 ist es möglich, die Multiplikation direkt mit zwei 16-Bit-Zahlen auszuführen. Wenn die Operation beendet ist, steht das Ergebnis in zwei 16-Bit-Registern. Dies ist notwendig, weil $2^{16} \times 2^{16}$ gleich 2^{32} ist. Deshalb wird ein 32-Bit-Register benötigt, um das bei der Multiplikation zweier 16-Bit-Zahlen größtmögliche Ergebnis zu speichern.

Beim 8086/8088 wird eine der zu multiplizierenden Zahlen im AX-Register gespeichert. Das doppeltlange (32-Bit-)Ergebnis wird in den Registern AX und DX abgelegt, mit den 16 signifikantesten Bits im DX-Register.

Wenn die Multiplikation eine 8-Bit-Operation sein soll, muß eine der Zahlen im AL-Register stehen. Das 16-Bit-Ergebnis wird in das AX-Register zurückgegeben, mit dem signifikantesten Byte in AH.

Abb. 5.9 stellt ein Programm zur Multiplikation zweier 16-Bit-Zahlen vor, die in den Speicherzellen ADR3 und ADR4 gespeichert sind.

```
                    1;
                    2;8086/8088 PROGRAMM ZUR MULTIPLIKATION ZWEIER
                    3;16-BIT-ZAHLEN. DIE ZAHLEN SIND
                    4;2345 x 5378 = 12 611 410 = 00C06F52H
                    5;
                    6;ORG 250H
                    7;
                    8 ADR3          EQU 300H
                    9 ADR4          EQU ADR3 + 2
                   10;
0250 B82909         11 MOV AX, LDA#2345   ;2345 IN DAS AX-REGISTER
0253 B90215         12 MOV CX,#5378
0256 F7E1           13 MUL CX             ;MULTIPLIZEIRE AX x (CX)5378
0258 A30003         14 MOV ADR3,AX        ;SPEICHERE UNTERE 16 BITS IN ADR3
025B 89160203       15 MOV ADR4,DX        ;SPEICHERE OBERE ;( BITS IN ADR4
                    16;
```

Abb. 5.9: Dies ist ein assembliertes 8086-8088-Programm zur Multiplikation zweier 16-Bit-Zahlen

Binäre Division

Lassen Sie uns unsere Erörterung der binären Division mit einer Untersuchung der dezimalen Division beginnen. Wir wollen 254 durch 12 dividieren:

```
(Dividend)  (Divisor)    (Quotient)
    254     :  12        =  21
  - 24
    14
  - 12
     2      (Rest)
```

Bei der Untersuchung dieses Problems sehen wir, daß die Division durch Subtraktion des größtmöglichen Vielfachen des Divisors von den äußerst linken Ziffern des Dividenden durchgeführt wird. Dies erzeugt den neuen Dividenden 14. Der Multiplikator des Divisors wird nun die zweite Ziffer des Quotienten. Der Rest schließlich ist des Ergebnis der letztmöglichen Subtraktion.

Um das größte Vielfache des Divisors zu finden, das von dem Dividenden subtrahiert werden kann, müssen wir Probevergleiche und Subtraktionen durchführen. Es sollte bemerkt werden, daß bei der Bestimmung der ersten Ziffer des Quotienten die tatsächliche Zahl 20 ist, nicht 2; und die Zahl, die von dem Dividenden subtrahiert wird ist 240, nicht 24. Durch Weglassen der Nullen sind wir in der Lage, die Notation passend zu machen, aber wir dürfen nicht aus den Augen verlieren, was in diesem Prozeß tatsächlich passiert.

Die binäre Division wird auf die gleiche Weise durchgeführt, wie man in dem folgenden Beispiel sehen kann:

```
(Dividend)  (Divisor)    (Quotient)
    1010    :  11        =  011
  - 11
    100
  - 11
      1     (Rest)
```

Dieses Beispiel ist derselben Vorschrift gefolgt, wie das vorher vorgestellte dezimale Beispiel. Wie man sieht, führt die Division zu einem Quotienten und einem Rest.

Division mit dem 8086/8088

Der 8086/8088 verfügt über einen Divisionsbefehl, der eine vollständige Divisionsoperation für uns ausführt. Der Dividend für die CPU ist in den

Registern AX und DX gespeichert. Das bedeutet, daß der Dividend eine 32-Bit-Größe ist, mit den 16 signifikantesten Bits im DX-Register.

Bei dieser Division ist der Divisor eine 16-Bit-Zahl. Wenn die Operation beendet ist, wird der 16-Bit-Quotient im AX-Register gespeichert. Ein 16-Bit-Rest wird in das DX-Register zurückgegeben.

Wenn die Division mit Byte-Größen durchgeführt wird, wird der 16-Bit-Dividend im AX-Register gespeichert. Am Schluß der Operation wird der 8-Bit-Quotient in das AL-Register zurückgegeben, während der Rest im AH-Register gespeichert wird.

Abb. 5.10 zeigt ein 8086/8088-Programm, das zwei Zahlen dividiert und den Quotienten und den Rest im Speicher ablegt.

```
                    1;
                    2;EIN 8086 8088-PROGRAMM ZUR DIVISION ZWEIER ZAHLEN
                    3;DAS PROBLEM IST, 450 DURCH 20 ZU DIVIDIEREN
                    4;DIES IST GLEICH 22 MIT EINEM REST = 10
                    5;
                    6;ORG 250H
                    7;
                    8 ADR3      EQU 300H          ;2 BYTES FÜR DEN QUOTIENTEN
                    9 ADR4      EQU ADR3 + 2      ;2 BYTES FUR DEN REST
                   10;
0250 31D2           11 XOR DX,DX            ;LÖSCHE DAS DX-REGISTER
0252 B8C201         12 MOV AX,#450          ;450 WIRD IN DAS AX-REGISTER GELADEN
0255 B91600         13 MOV CX,#22           ;22 WIRD IN DAS CX-REGISTER GELADEN
0258 F7F1           14 DIV CX               ;DIVIDIERE DXAX DURCH CX(22)
025A A30003         15 MOV ADR4,AX          ;SPEICHERE QUOTIENTEN IN SPEICHER
025D 89160203       16 MOV ADR4,DX          ;SPEICHERE REST IM SPEICHER
                    17;
```

Abb. 5.10: Dies ist ein assembliertes 8086-8088-Programm zur Division zweier 16-Bit-Zahlen

UNTERPROGRAMME

In diesem Abschnitt wollen wir untersuchen, wie der 8086 und der 8088 mit Unterprogrammen arbeiten. Wir beginnen mit einer allgemeinen Definition eines Unterprogramms; dann fahren wir fort mit den Details über die Benutzung eines Unterprogramms mit der CPU.
Im Prinzip ist ein *Unterprogramm* einfach ein Block oder Abschnitt von Befehlen, der vom Programmierer benannt wird. Ein Unterprogramm wird ausgeführt, wenn das Hauptprogramm einen besonderen Aufruf-

Befehl (CALL) verarbeitet. Das Unterprogramm wird dann durch einen besonderen Befehl, Rücksprung (RETURN) genannt, beendet. Lassen Sie uns nun die Benutzung eines Unterprogramms veranschaulichen, um seinen Wert aufzuzeigen.

Abb. 5.11 zeigt, wie ein Unterprogramm benutzt wird. Das Hauptprogramm erscheint links in der Abbildung, und das Unterprogramm erscheint symbolisch rechts. Lassen Sie uns untersuchen, wie das Unterprogramm arbeitet.

In diesem Programm werden die Zeilen des Hauptprogramms nacheinander ausgeführt, bis ein CALL SUBROUTINE-Befehl auftritt. Die Aus-

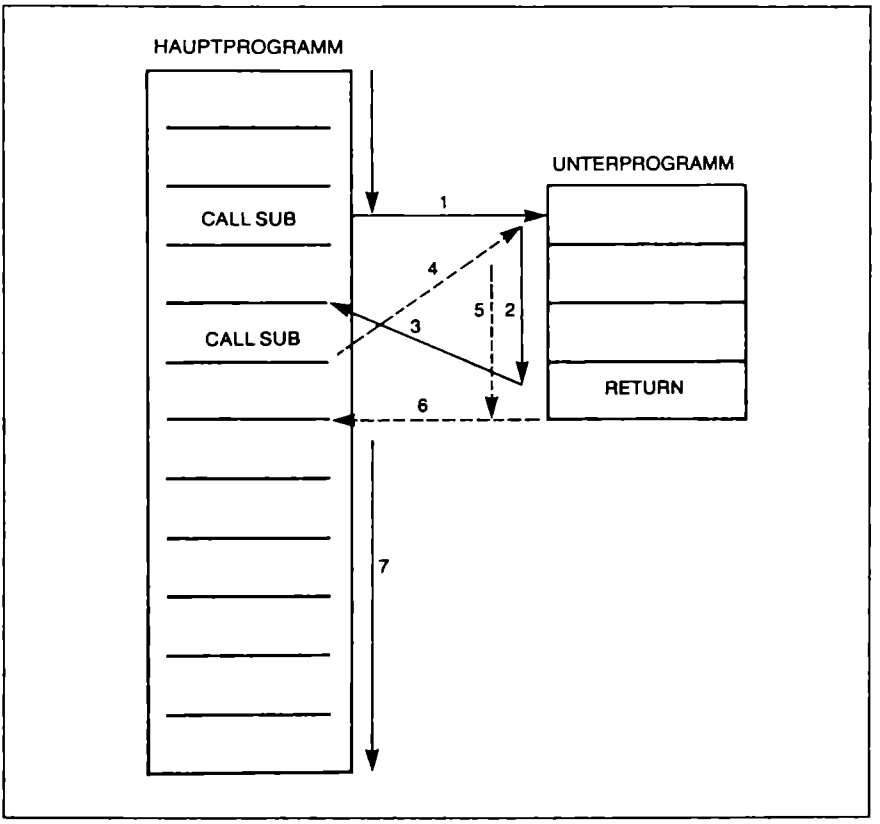


Abb. 5.11: Ein Blockdiagramm, das zeigt, wie ein Unterprogramm von verschiedenen Positionen im Hauptprogramm aus aufgerufen werden kann. Die durchgezogenen Pfeile 1, 2 und 3 zeigen den Ausführungspfad für das erste CALL SUB; die unterbrochenen Pfeile 4, 5 und 6 zeigen den Pfad für das zweite CALL SUB

führung dieses Befehls hat dann einen Übergang in den Unterprogrammabschnitt des Programms zur Folge. (Also ist der nächste nach dem CALL SUBROUTINE auszuführende Befehl der erste Befehl in dem Unterprogramm.) Dies wird in Abb. 5.11 durch Pfeil 1 veranschaulicht. Der Unterprogrammabschnitt, den wir in diesem Beispiel als Unterprogramm verwenden, wird wie jedes andere Programm ausgeführt, wie Pfeil 2 anzeigt. Mit anderen Worten: Das Unterprogramm enthält keine CALL SUBROUTINE-Befehle (wie später beschrieben).

Der letzte Befehl des Unterprogramms ist ein RETURN. Dieser besondere Befehl veranlaßt die CPU, in das Hauptprogramm zurückzukehren. Nach der Rückkehr führt sie den Befehl aus, der unmittelbar auf den CALL SUBROUTINE-Befehl folgt, der die CPU ursprünglich gezwungen hat, in den Unterprogrammabschnitt des Programms zu gehen. Dies wird durch Pfeil 3 in Abb. 5.11 gezeigt.

Später erscheint ein zweiter CALL SUBROUTINE-Befehl im Hauptteil des Programms. Ein neuer Übergang erfolgt, wie durch Pfeil 4 gezeigt wird. Dies bedeutet, daß der Rumpf des Unterprogramms gemäß dem CALL SUBROUTINE-Befehl wieder ausgeführt wird.

Immer wenn ein RETURN in einem Unterprogramm auftritt, erfolgt ein Rücksprung zu dem Befehl, der auf den gerade ausgeführten CALL SUBROUTINE-Befehl folgt. Dies wird durch Pfeil 6 veranschaulicht. Nach dem Rücksprung in das Hauptprogramm geht die Ausführung normal weiter, wie Pfeil 7 veranschaulicht.

Der Effekt der beiden Befehle CALL SUBROUTINE und RETURN sollte nun klar sein. Was ist der Nutzen eines Unterprogramms? Der wesentliche Nutzen eines Unterprogramms ist der, daß es von jedem beliebigen Punkt des Hauptprogramms aus aufgerufen werden kann, und daß die Befehle in einem Unterprogramm wiederholt verwendet werden können, ohne neu geschrieben werden zu müssen. Ein Vorteil dieser Methode ist die Einsparung von Speicherplatz, weil das Unterprogramm nicht jedesmal neu aufgeschrieben werden muß. Ein anderer Vorteil liegt darin, daß der Programmierer ein spezielles Unterprogramm nur einmal zu entwerfen braucht, während es wiederholt verwendet werden kann. Dies ist eine bedeutende Vereinfachung des Programmentwurfsprozesses.

Implementierung des Unterprogramm-Mechanismus

Nun wollen wir erörtern, wie die beiden Befehle CALL SUBROUTINE und RETURN innerhalb des Mikroprozessors implementiert werden. Zuerst behandeln wir CALL SUBROUTINE. Der CALL SUBROUTINE-Befehl bewirkt, daß eine neue Adresse in das IP-Register der CPU geschrieben wird. Rufen Sie sich ins Gedächtnis zurück, daß das IP-Register die Speicheradresse des nächsten auszuführenden Befehls enthält.

Mit anderen Worten: Die Startadresse des Unterprogramms wird in das IP-Register geladen. Aber ist das genug?

Um diese Frage zu beantworten, lassen Sie uns den anderen Befehl betrachten: RETURN. Dieser Befehl verursacht einen Rücksprung zu dem Hauptprogrammbefehl, der dem CALL SUBROUTINE folgt. Eine solche Handlung ist aber nur möglich, wenn die Speicheradresse des Befehls, der dem CALL SUBROUTINE folgt, irgendwo erhalten wird.

Das nächste Problem hängt mit der Rettung dieser Rücksprungadresse zusammen: Sie muß immer an eine Stelle gerettet werden, an der sie nicht gelöscht wird.

Wenn die CPU auf einen CALL SUBROUTINE-Befehl stößt, wird der Wert des IP-Registers automatisch gerettet. Zu diesem Zeitpunkt ist das IP-Register gleich der Offset-Adresse des nächsten auszuführenden Befehls. In diesem Fall ist das die Speicheradresse, die direkt dem CALL SUBROUTINE-Befehl folgt.

Als nächstes beginnt die CPU die Ausführung an der Adresse des Unterprogramms. Wenn der RETURN-Befehl in dem Unterprogramm ausgeführt wird, wird der gerettete Adreßwert aus der Speicherzelle, in die er gerettet wurde, nach IP zurückgebracht. Das Retten und Umspeichern der Adresse des Befehls nach dem CALL SUBROUTINE-Befehl wird automatisch von der CPU ausgeführt. Der CALL-Befehl rettet die Adresse, und der RETURN-Befehl speichert die Adresse um.

Wir haben vorhin festgestellt, daß die Adresse an eine bestimmte Stelle im Speicher gerettet wird. Die Frage ist, „Wohin im Speicher wird die Adresse gerettet?". Sie wird an die Stelle im Speicher gerettet, die als Stapel bezeichnet wird. Der Stapel ist ein besonderer Speicherabschnitt, der für gewisse CPU-Operationen reserviert ist. Wir werden diese Operationen in späteren Kapiteln dieses Buches angeben.

In Kapitel 4 haben wir die Befehle PUSH und POP vorgestellt, die den Systemstapel benutzen. Wenn ein CALL SUBROUTINE-Befehl auftritt, wird die Adresse des nächsten auszuführenden Befehls auf den Stapel „gePUSHt", dorthin gerettet. Wenn die CPU auf einen RETURN-Befehl trifft, wird die gerettete Adresse automatisch vom Stapel „gePOPt", d.h. zurückgeholt. Diese vom Stapel genommene Adresse wird dann als die Adresse für den nächsten Befehl genommen.

Intrasegment-Call

Es gibt zwei Haupttypen von Unterprogrammaufrufen: INTRASEGMENT- und INTERSEGMENT-Aufruf. Wir wollen sie nun untersuchen.

Der Intrasegment-CALL-Befehl veranlaßt die CPU, ein Unterprogramm auszuführen, das in demselben 64K-Speicherblock wie das aufrufende Programm steht. Wenn der 8086/8088 auf diesen Typ von CALL stößt,

wird das IP-Register auf den Stapel gebracht. Wenn ein RETURN-Befehl auftritt, wird der IP-Registerwert vom Stapel heruntergenommen und wieder in das IP-Register der CPU gebracht. Siehe Abb. 5.12.

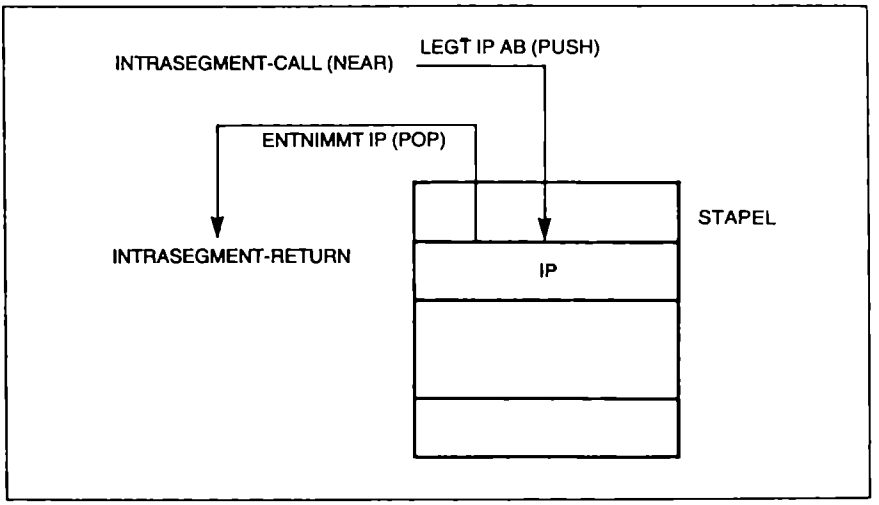


Abb. 5.12: Ein Intrasegment- (innerhalb eines 64K-Blocks von Daten) CALL legt das IP-Register auf dem Systemstapel ab. Das entsprechende Intrasegment-RETURN entnimmt das IP-Register wieder dem Stapel

Intersegment-Call

Dieses CALL veranlaßt den 8086/8088 ein Unterprogramm auszuführen, das außerhalb des 64K-Codesegmentblocks steht. In diesem Fall werden beide, das CS- (code segment, Codesegment-)Register und das IP-Register auf dem Stapel abgelegt. Wenn ein Rücksprungbefehl auftritt, werden die beiden geretteten Werte, CS- und IP-Wert, dem Stapel entnommen und in die internen Register der CPU geladen. Dies wird in Blockdiagrammform in Abb. 5.13 gezeigt.

CALL-RETURN-UNSTIMMIGKEIT

An den vorangegangenen Beispielen von Intrasegment- und Intersegmentaufrufen können wir sehen, daß der Rücksprungbefehl entweder das CS-Register und das IP-Register oder nur das IP-Register dem Stapel entnehmen muß. Deshalb muß es mehr als eine Art von Rücksprungbefehl geben. So ist es tatsächlich. Rufen Sie sich aus Kapitel 4 ins Gedächtnis zurück, daß es Intersegment- und Intrasegment-RETURNs gibt.

Dies bedeutet, daß ein Unterprogramm immer entweder als ein Intrasegment oder als ein Intersegmentprogramm verwendet werden muß. Abb. 5.14 zeigt eine CALL-RETURN-Unstimmigkeit.

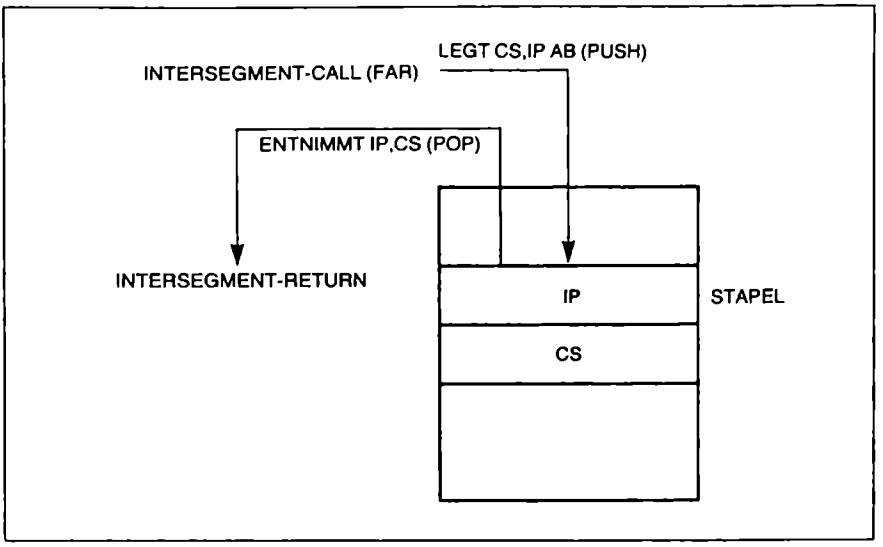


Abb. 5.13: Ein Intersegment- (außerhalb eines 64K-Blocks von Daten) CALL legt beide, das CS- und das IP-Register, auf dem Stapel ab. Ein Intersegment-RETURN entnimmt diese beiden Register wieder dem Stapel

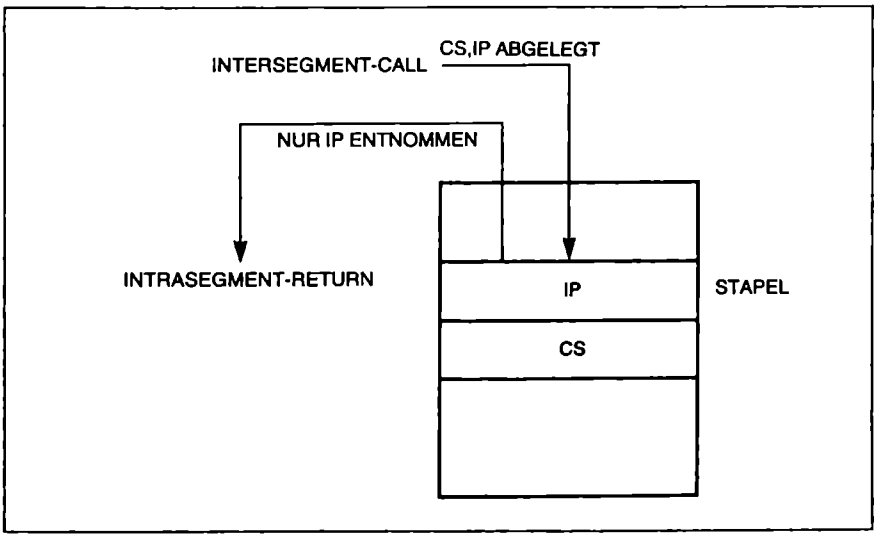


Abb. 5.14: Hier wird eine CALL-RETURN-Unstimmigkeit dargestellt. In diesem Programm legt ein Intersegment-CALL das CS- und das IP-Register auf dem Stapel ab. Ein Intrasegment-RETURN entnimmt nur das IP-Register. Dieses Vorgehen führt zu einer CALL-RETURN-Unstimmigkeit, und das Programm stürzt üblicherweise ab

ZUSAMMENFASSUNG

In diesem Kapitel haben wir einige grundlegende Programmierkonzepte vorgestellt. Diese umfaßten binäre Addition, Subtraktion, Multiplikation und Division ebenso wie BCD-Arithmetik. Wir haben mehrere Programme geschrieben und erklärt, wie der 8086/8088 diese Konzepte realisiert.

Gegen Ende des Kapitels haben wir den Nutzen von Unterprogrammen in der Programmierung erörtert. Wir haben untersucht, wie der 8086/8088 Unterprogrammaufrufe und -rücksprünge implementiert, und wir haben einen Blick auf Intersegment- und Intrasegmentaufrufe und -rücksprünge geworfen. Wir werden diese Information als nützlich ansehen, wenn wir verschiedene Programmieranwendungen für den 8086/8088 in den folgenden Kapiteln erörtern.

Kapitel 6

Unterbrechungen beim 8086/8088

In diesem Kapitel erörtern wir das allgemeine Thema der Unterbrechungen (Interrupts) beim 8086/8088. Zuerst stellen wir das Konzept einer Unterbrechung vor. Dann zeigen wir, wie jeder der verschiedenen Unterbrechungstypen vom 8086 und 8088 verarbeitet wird. Zusätzlich untersuchen wir nützliche Softwarebeispiele und zeigen auf, was tatsächlich während einer Unterbrechungsoperation in der CPU geschieht.

WAS IST EINE UNTERBRECHUNG?

Ein Beispiel für eine Unterbrechung wird am besten durch die folgende Szenerie gegeben. Stellen Sie sich dies vor:

Sie unterhalten sich mit einer anderen Person. Eine weitere Person tritt auf und sagt Ihren Namen – fordert damit Ihre Aufmerksamkeit. Das Folgende ist eine Liste der Möglichkeiten, auf diese externe Anfrage zu antworten:

1. Sie können die zweite Person vollständig ignorieren und Ihre Unterhaltung mit der ersten Person fortsetzen.
2. Sie können einen geeigneten Einschnitt in ihrer Unterhaltung abwarten, und dann die Aufmerksamkeit der zweiten Person zuwenden.
3. Sie können Ihre Unterhaltung mit der ersten Person sofort beenden und anfangen, mit der zweiten Person zu sprechen.

In jedem Fall ist es wahrscheinlich, daß Sie, wenn Sie aufgehört haben, mit der zweiten Person zu sprechen, Ihre Unterhaltung mit der ersten Person fortsetzen möchten.

Obwohl diese Szenerie vereinfachend scheinen mag, entspricht sie genau dem Konzept einer Unterbrechung innerhalb eines Mikroprozessorsystems. Stellen Sie sie sich in dieser Art vor. Nehmen Sie an, Sie selbst wären die 8086/8088-CPU. Stellen Sie sich die erste Person als das laufende Hauptprogramm und die zweite als eine externe Unterbrechungs-

anforderung vor (so wie eine Hardwarekomponente des Systems, die die Aufmerksamkeit der CPU erwartet). Die 8086/8088-CPU muß dieser Anforderung in irgendeiner Form nachkommen. Es gibt verschiedene Arten, das zu tun. Die drei oben aufgezählten Möglichkeiten sind die üblichsten.

Nach dieser allgemeinen Einführung über Unterbrechungen wollen wir uns nun auf die Details der Unterbrechungen beim 8086/8088 konzentrieren.

Woher kommen die externen Unterbrechungsanforderungen?

Ein Mikroprozessorsystem kann viele verschiedene Hardwarekomponenten umfassen: einen Drucker, ein Platten- oder Diskettenlaufwerk, einen Bildschirm, einen Zeitgeber, eine Kontrolleinrichtung oder einen Digital-Analog-Wandler (DAC) – um nur einige zu nennen. Die meisten dieser externen Hardwarekomponenten brauchen die Aufmerksamkeit der CPU nur zu bestimmten Zeiten. Zu anderen Zeiten funktionieren sie selbständig.

Lassen Sie uns zum Beispiel annehmen, Ihr System hat eine Uhr als externe Hardwareeinrichtung, die dazu entworfen ist, die Tageszeit auf dem CRT-Schirm anzuzeigen. Mit anderen Worten: Wenn der Schirm betrachtet wird, fordert die Zeithardware, daß die CPU die Zeit einmal pro Sekunde liest und auf den CRT-Schirm schreibt. Zu allen anderen Zeiten steht die CPU zur Ausführung ihrer anderen Aufgaben zur Verfügung.

Der wichtigste Punkt ist hier, daß die externe Hardware der Uhr die Aufmerksamkeit der CPU nicht über *die gesamte Zeit* benötigt. Die Zeithardware beansprucht die CPU zum Ablesen der Zeit nur, wenn es erforderlich ist. Eine Art, dies zu erreichen, geht über das Unterbrechungssystem der CPU, wobei die CPU einmal pro Sekunde eine externe Unterbrechungsanforderung von der Zeithardware erhält. Dann bricht die CPU das, was sie gerade tut, ab und liest die Uhrzeit. Nachdem die Uhr abgelesen und die Zeit dargestellt wurde, nimmt die CPU die Ausführung des Programms, das sie vor der Unterbrechung abgearbeitet hat, wieder auf.

Dieses einfache Beispiel für eine Unterbrechung sollte die Ausgangsfrage beantworten helfen, woher die Unterbrechungsanforderungen kommen. Die Antwort ist, daß sie von der Mikroprozessorhardware kommen.

Nichtmaskierbare Unterbrechungen (NMI)

Wie in Abb. 6.1 dargestellt, gibt es drei Hauptunterbrechungseingabeleitungen auf den 8086/8088 Mikroprozessoren: INTR, NMI und RESET. Jede dieser Eingabeleitungen kann eine externe Unterbrechungsanforderung bewirken. In diesem Abschnitt wollen wir die NMI (non maskable interrupt) oder nichtmaskierbare Unterbrechungsanforderung behan-

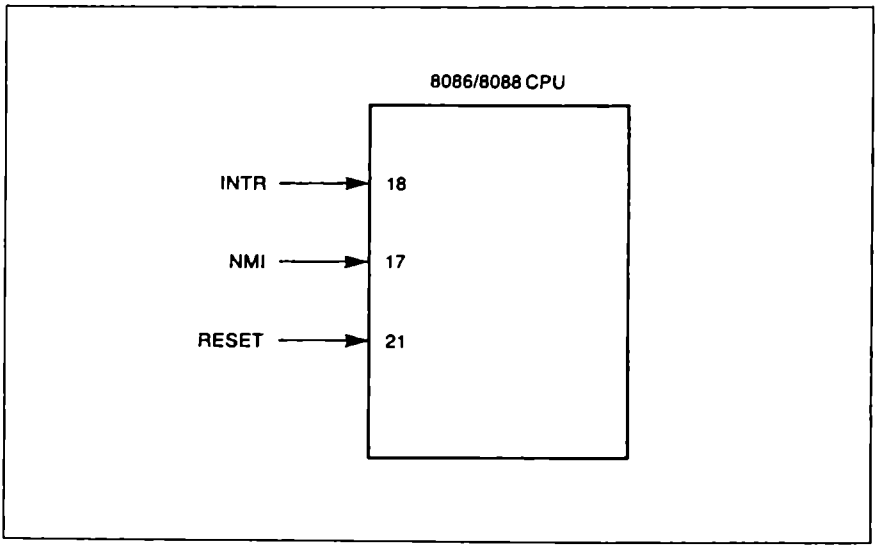


Abb. 6.1: Dieses Blockdiagramm zeigt die INTR-, NMI- und RESET-Unterbrechungseingänge des 8086/8088

deln. Wir werden mehrere wichtige Punkte, die die 8086/8088 Unterbrechungen betreffen, ausführlich erörtern. Viele dieser Punkte sind anderen Typen von Unterbrechungseingaben gemeinsam. Wir beginnen unsere Diskussion mit der nichtmaskierbaren Unterbrechung, weil sie nur auf eine Art arbeitet.

Der Ausdruck „nichtmaskierbare Unterbrechung" bedeutet, daß diese Unterbrechung immer so schnell wie elektrisch möglich von dem 8086/8088 erkannt werden muß. (Dies war die dritte Möglichkeit in unserer vorangegangenen Liste.) Da der 8086/8088 eine nichtmaskierbare Unterbrechung hat, folgt, daß er auch eine maskierbare Unterbrechungseingabe haben muß. Dies ist die INTR-Eingabe. Diese Eingabe werden wir später in diesem Kapitel behandeln.

Lassen Sie uns annehmen, daß die NMI-Unterbrechungseingabe extern verursacht wurde. Die nächste Frage ist dann: „Wann wird die NMI-Eingabe erkannt?" Im allgemeinen wird sie am Ende des laufenden Befehlszyklus erkannt; das heißt, wenn der gerade in der Ausführung befindliche Befehl beendet ist. Beachten Sie, daß es einige Befehle auf dem 8086/8088 gibt, die eine recht lange Ausführungszeit beanspruchen können (z. B. der Divisions- und der Multiplikationsbefehl). Die Zeit, die die CPU braucht, um eine Unterbrechungsanforderung zu beantworten, wird als *Unterbrechungslatenz* bezeichnet. Der tatsächliche Zeitaufwand hängt davon ab, wieviele Zeittakte der laufende Befehl noch benötigt.

CPU-TÄTIGKEIT WÄHREND DER NMI-ANFORDERUNG

Während einer NMI-Operation werden als erstes die Kennzeichenbits (flags) in den Systemstapelbereich gebracht. Diese Handlung rettet die aktuellen Werte aller Systemkennzeichen. Als nächstes löscht die CPU das IF (das Unterbrechungszulassungskennzeichen) und verhindert somit jede Unterbrechung, die an der INTR-Eingabeleitung auftritt. Wenn dieses Kennzeichenbit gelöscht ist, werden alle Unterbrechungsanforderungen von der INTR-Eingabe ignoriert.

Anschließend wird ein besonderes Kennzeichenbit auf dem 8086/8088, TF (Trap-Kennzeichenbit) genannt, ebenfalls gelöscht. Dieses Kennzeichenbit wird benutzt, wenn sich der 8086/8088 in einem Einzelschrittmodus befindet. Es zu löschen bestimmt daher, daß der 8086/8088 nicht länger im Einzelschrittmodus benutzt wird.

Die nächste von der CPU vorgenommene Handlung ist, das CS- und das IP-Register auf den Systemstapel zu bringen. An diesem Punkt der Behandlung des NMI durch die CPU erscheint der Systemstapel wie in Abb. 6.2 gezeigt.

Schließlich lädt die CPU das IP-Register mit dem 16-Bit-Wert, der an der Speicheradresse 00008H steht. Das CS-Register wird dann mit dem 16-Bit-Wert, der an der Adresse 0000AH steht, geladen. Nun wird eine neue Speicheradresse erzeugt, die die neu geladenen CS- und IP-Register benutzt. Schließlich beginnt die CPU mit der Ausführung des Codes, der in dieser neuen Speicheradresse steht.

Hier sind in einem Überblick die Schritte, die ausgeführt werden, wenn die CPU eine NMI-Anforderung abarbeitet:

1. Das Kennzeichenregister wird auf den Systemstapel gebracht.
2. Die INTR-Eingaben werden verhindert.
3. TF wird gelöscht, es kann kein Einzelschritt mehr auftreten.
4. Das CS-Register wird auf den Systemstapel gebracht.
5. Das IP-Register wird auf den Systemstapel gebracht.
6. Das IP-Register wird mit dem 16-Bit-Wert an der Speicheradresse 00008H geladen.
7. Das CS-Register wird mit dem 16-Bit-Wert an der Speicheradresse 0000AH geladen.
8. Die CPU holt sich den nächsten Befehl aus der 20-Bit-Adresse, die von den neuen CS- und IP-Registern erzeugt wurde.

Nun führt die CPU die Unterbrechungsserviceroutine für eine NMI-Unterbrechungsanforderung aus.

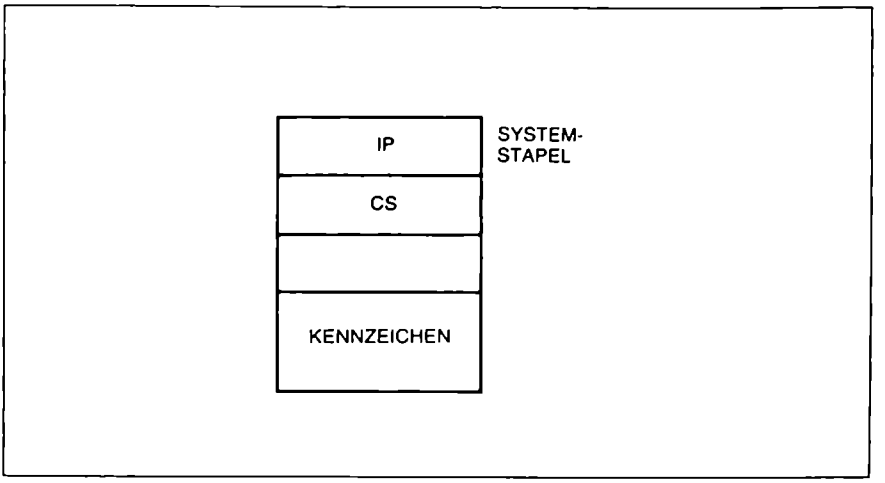


Abb. 6.2: So sieht der Stapel aus, nachdem die Kennzeichen, das CS- und das IP-Register während einer Unterbrechungsbehandlung auf dem Stapel abgelegt wurden

DIE UNTERBRECHUNGSTABELLE

Bevor wir mit unserer Diskussion über die INTR-Eingabe fortfahren, lassen Sie uns das Konzept der Unterbrechungstabelle untersuchen. In unserer Erklärung der NMI-Unterbrechung haben wir festgestellt, daß die Adressen 00008H und 0000AH zur Speicherung des IP- und des CS-Registers verwendet werden. Die in diesen Registern gespeicherten Werte werden zur Erzeugung der neuen 20-Bit-Adresse verwendet. Diese Adreßwerte wurden aus einer Folge von Adressen, die als Unterbrechungstabelle (interrupt table) bezeichnet wird, gewonnen. Abb. 6.3 zeigt ein Diagramm der Unterbrechungstabelle für den 8086/8088. Diese Unterbrechungstabelle belegt die ersten 1024 Bytes des Systemspeichers, das heißt die Speicheradressen 00000-003FF. Es gibt 256 verschiedene Arten von Unterbrechungen, die vom 8086/8088 verarbeitet werden können. Später in diesem Kapitel erörtern wir das Thema der Unterbrechungsarten detaillierter. Sie werden lernen, daß jede Art mit einer Markierung von Typ 0 – Typ 255 versehen wird.

Für jede Art der Unterbrechung (von 0–255) sind 4 Speicherbytes in der Unterbrechungstabelle reserviert. Diese vier Bytes werden zur Speicherung der 16-Bit-CS- und IP-Register benötigt, die zur Erzeugung der 20-Bit-Adresse der Unterbrechungsserviceroutine verwendet werden. Die ersten zwei der vier Bytes enthalten den Wert des IP-Registers. Die letzten zwei Bytes enthalten den Wert des CS-Registers wie Abb. 6.4 zeigt.

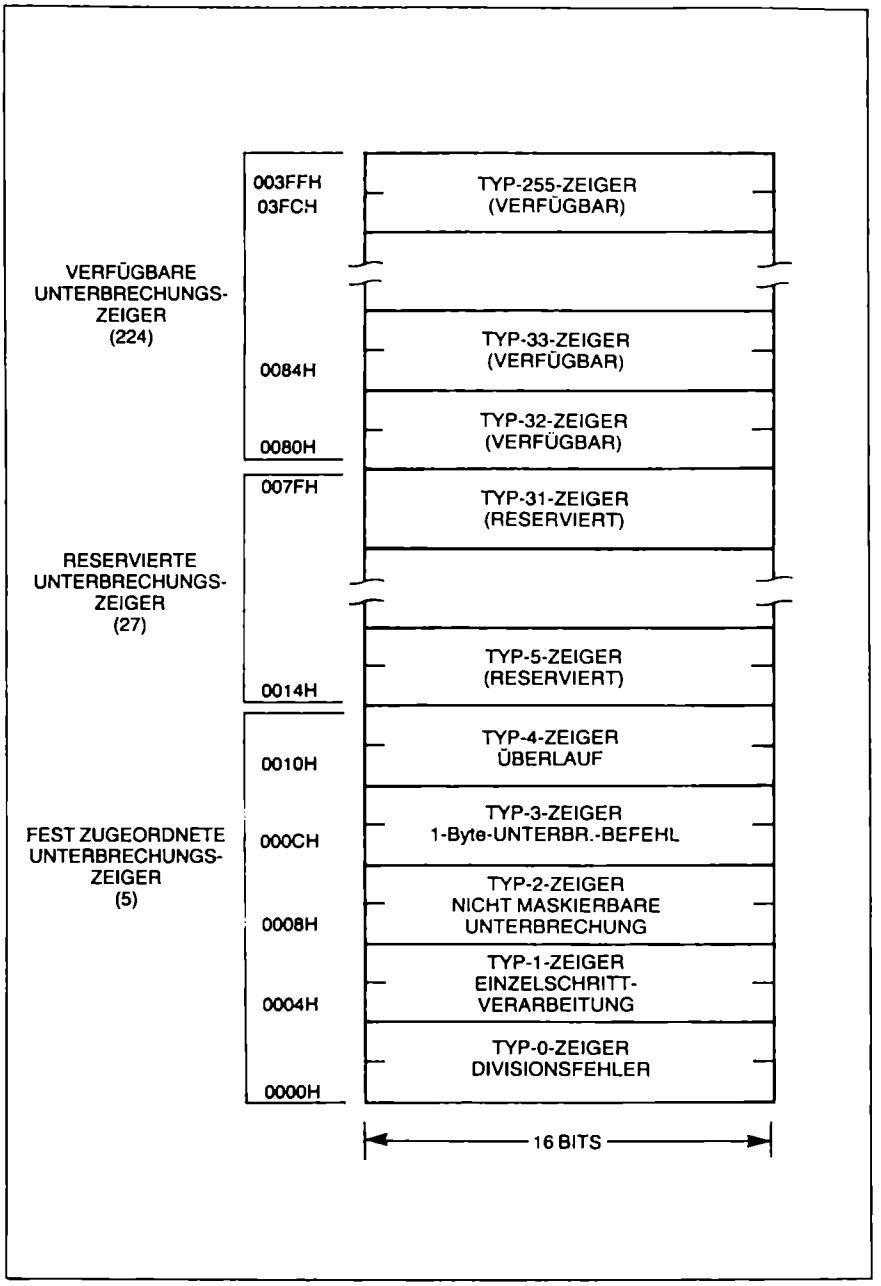


Abb. 6.3: Die Unterbrechungstabelle für den 8086/8088. Jeder Unterbrechungstyp benötigt 4 Bytes Speicherplatz

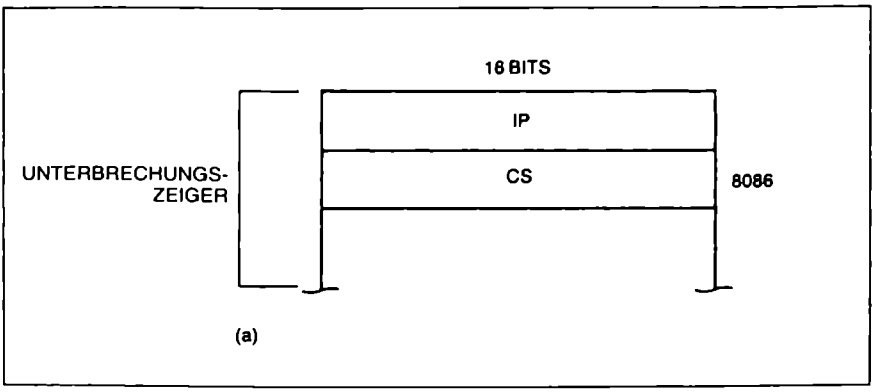


Abb. 6.4a: Der Unterbrechungszeiger für den 8086 benötigt zwei 16-Bit-Wörter. Die niedrigere Adresse ist gleich dem IP-Register, die höhere Adresse ist gleich dem CS-Register

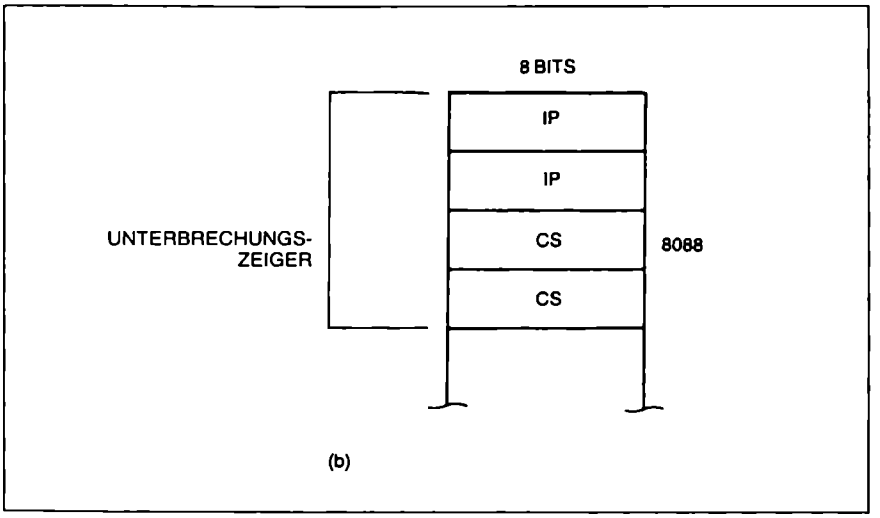


Abb. 6.4b: Der Unterbrechungszeiger für den 8088 benötigt vier aufeinanderfolgende Bytes des Speichers.

Reservierte Unterbrechungsarten

Einige der 256 möglichen Unterbrechungsarten sind für spezielle Anwendungen reserviert. Das Folgende ist eine Liste der Unterbrechungen von Typ 0 – Typ 4 und ihrer besonderen Bedeutung in dem System:

Typ 0 Reserviert für die Division durch 0.

Typ 1 Reserviert für Einzelschrittverarbeitung.

Typ 2 Reserviert für NMI-Unterbrechungen.

Typ 3 Reserviert für einen 1-Byte-Software-Unterbrechungsbefehl.

Typ 4 Reserviert für eine Überlaufsunterbrechung mit Vorzeichen.

Die Firma Intel (die den 8086/8088 entwickelt hat) hat die Unterbrechungsarten von Typ 5-31 (Speicheradressen 00014H – 0007FH) für spätere Intelprodukte reserviert. Mit anderen Worten: Wenn Sie eine Unterbrechung vom Typ 5-31 benutzen, ist Ihr System möglicherweise nicht in der Lage, ein späteres von Intel entwickeltes Produkt zu verwenden.

INTR-EINGABE

Lassen Sie uns nun die INTR-Eingabe behandeln. Diese Unterbrechungseingabe kann durch Software in und außer Betrieb gesetzt werden. Wenn sie außer Betrieb gesetzt ist, wird die CPU keine externe Anforderung beachten. In Kapitel 4 haben wir den CLI-Befehl (die Unterbrechungssperre) behandelt. Dieser Softwarebefehl wird benutzt, um die INTR-Eingabe außer Betrieb zu setzen.

Wir können die INTR-Eingabe der 8086/8088-CPU durch Verwendung des STI-Befehls in Betrieb setzen (Unterbrechungsfreigabe). Lassen Sie uns zum Beispiel annehmen, daß die INTR-Eingabe in Betrieb gesetzt worden ist, und eine externe Anforderung besteht. Im folgenden wird erklärt, wie die CPU diese Arten von Anforderungen verarbeitet.

Wenn die INTR-Eingabe von der CPU beachtet worden ist, wird ein externes Signal, *Unterbrechungsbestätigung* genannt, erzeugt. Dieses Signal erlaubt der Systemhardware, einen 8-Bit-Wert, der *Unterbrechungstyp* genannt wird, auf den Systemdatenbus zu legen. Rufen Sie sich ins Gedächtnis zurück, daß es 256 verschiedene Typen gibt.

Als nächstes liest die CPU den Unterbrechungstyp vom Datenbus und berechnet die Adresse des Typs in der Unterbrechungstabelle. Dies geschieht durch Multiplikation der vom Systemdatenbus gelesenen Typnummer mit 4. Lassen Sie uns zum Beispiel annehmen, die externe Hardware legt eine 2FH auf den Systemdatenbus, wenn die Unterbrechungsbestätigung festgestellt wird. 2FH ist die Typnummer. Die Adresse für diese Typnummer in der Unterbrechungstabelle ist 4 x 2FH, das ist gleich 000BCH. Als nächstes werden die Kennzeichenbits auf den Systemstapel gebracht. Anschließend werden die Unterbrechungen gesperrt.

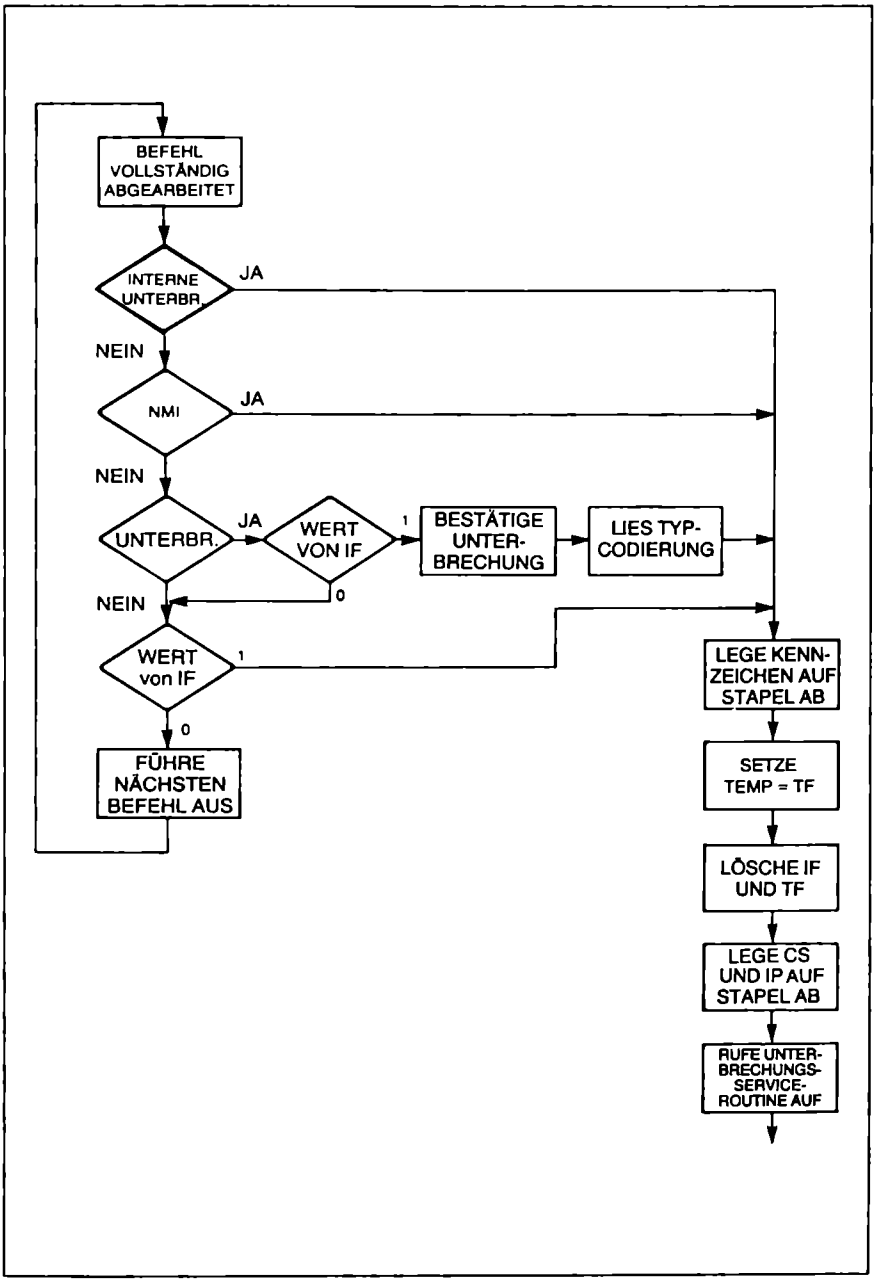


Abb. 6.5: Dieses Flußdiagramm zeigt die Ereignisse, die während der Behandlung von Unterbrechungen mit dem 8086/8088 auftreten.

Die CPU bringt nun das IP- und das CS-Register auf den Systemstapel. Danach lädt die CPU neue Werte für das CS- und das IP-Register aus den Speicherzellen 000BCH – 000BFH. Wenn diese Register geladen sind, holt der Mikroprozessor den nächsten Befehl aus der 20-Bit-Adresse, die durch das neue CS- und IP-Register erzeugt wurde.

Lassen Sie uns nun auf die Liste von Handlungen zurückblicken, die die CPU bei der Verarbeitung einer INTR-Anforderung ausführt:

1. Ein externes Unterbrechungsbestätigungssignal wird erzeugt.
2. Ein Typcode (0-255) wird vom Systemdatenbus gelesen.
3. Das Register mit den Kennzeichenbits wird auf den Systemstapel gebracht.
4. Die Unterbrechungen und der Einzelschrittmodus werden gesperrt.
5. Das CS- und das IP-Register werden auf den Systemstapel gebracht.
6. Das IP-Register wird mit den Werten an den Speicheradressen (TYPE x 4) und (TYPE x 4)+1 geladen.
7. Das CS-Register wird mit den Werten an den Speicheradressen (TYPE x 4) +2 und (TYPE x 4)+3 geladen.
8. Die CPU holt den nächsten Befehl aus der 20-Bit-Speicheradresse, die durch die neuen Werte des CS- und des IP-Registers erzeugt wurde, die gerade aus der Unterbrechungstabelle geladen worden sind.

Abb. 6.5 zeigt in einem Flußdiagramm die Ereignisse, die während der Verarbeitung von Unterbrechungen mit dem 8086/8088 auftreten.

INTERNE UNTERBRECHUNGEN

Externe Unterbrechungen sind nur eine Art, Unterbrechungen mit dem 8086/8088 zu erzeugen. Eine andere Technik ist die Softwareunterbrechung. Unter Benutzung spezieller Befehle ist es möglich, die CPU in die Lage zu versetzen, auf irgendeinen Unterbrechungstyp (0-255) zu antworten. Hier eine Beschreibung, wie dies geschieht.

Ein Softwarebefehl, der eine Unterbrechung verursacht, ist INT (vorgestellt in Kapitel 4). Es gibt zwei Arten, diesen speziellen Befehl zu benutzen – entweder als 1-Byte-Befehl oder als 2-Byte-Befehl. Wenn Sie einen 1-Byte-INT-Befehl benutzen, wird er als CCH verschlüsselt.

Wenn die CPU diesen 1-Byte-Befehl ausführt, wird automatisch eine Typ-3-Unterbrechung erzeugt in genau der gleichen Art wie bei der INTR-Eingabe. Anders jedoch als die INTR-Eingabe kann die Softwareunterbrechung nicht gesperrt oder maskiert werden.

Nun wollen wir annehmen, daß wir die 2- Byte-Version des INT-Befehls benutzen. Das erste Byte würde als CDH codiert. Das zweite Byte wäre gleich einer 8-Bit-Zahl zwischen 0 und 255 (einschließlich). Diese Zahl definiert den Unterbrechungstyp, den Sie erzeugen möchten.

Lassen Sie uns zum Beispiel annehmen, Sie wollten einen Unterbrechungstyp 96H erzeugen. Der 2-Byte-INT-Befehl würde als CD96H verschlüsselt. Die hexadezimale Unterbrechungstabellenadresse wäre für IP gleich 4 x 96H, also 258 hexadezimal. Die Adressen 258H und 259H enthielten den IP-Registerwert für die Unterbrechungsserviceroutine. Die Adressen 25AH und 25BH enthielten den CS-Registerwert für die Unterbrechungsserviceroutine.

Unterbrechung nach einem Überlauf

Es gibt einen anderen Softwarebefehl, der eine automatische Erzeugung einer Typ-4-Unterbrechung bewirkt: der INTO-Befehl. Wenn das Überlaufkennzeichenbit (OF) gesetzt ist und dieser Befehl ausgeführt wird, wird eine interne Unterbrechung erzeugt. Wegen einer vollständigen Beschreibung eines Überlaufs schlagen Sie bitte in Kapitel 1 nach.

Rücksprung von einer Unterbrechung

Nun da wir gelernt haben, wie die CPU die Ausführung einer Unterbrechungsserviceroutine beginnt, lassen Sie uns erörtern, wie die CPU von der Serviceroutine zm Hauptprogramm zurückkehrt.

Wenn die CPU anfangs in die Serviceroutine einsteigt, werden die Kennzeichenbits, das CS- und das IP-Register gerettet. Dies ist im wesentlichen die Information, die gebraucht wird, um die CPU zu der Adresse, die sie vor der Unterbrechung enthielt, zurückzubringen.

Während der Ausführung der Unterbrechungsserviceroutine können Sie die Werte einiger interner Register verändern. Wenn es wichtig ist, die ursprünglichen Werte dieser Register zu retten, ist es vernünftig, sie auf dem Stapel abzulegen. Abbildung 6.6 zeigt einen typischen Start einer Unterbrechungsserviceroutine.

Ein anderer bemerkenswerter Punkt für das Einsteigen in eine Unterbrechungsserviceroutine ist der, daß alle Unterbrechungen gesperrt worden sind. Wenn Sie möchten, daß andere Unterbrechungen von der CPU während der Ausführung eines Interrupts beachtet werden, sollte der Unterbrechungsfreigabebefehl STI einer der ersten Befehle sein, die Sie in der Unterbrechungsserviceroutine unterbringen müssen. Dies wird durch den Programmabschnitt in Abbildung 6.7 dargestellt.

Wir wollen nun annehmen, wir haben die Unterbrechung vollständig abgearbeitet. Wie kehren wir nun zum Hauptprogramm zurück? Wir

```
;
; START DER UNTERBRECHUNGSSERVICEROUTINE
;
                PUSH AX         ;RETTE AX-REGISTER
                PUSH BX         ;RETTE BX-REGISTER
                PUSH CX         ;RETTE CX-REGISTER
                PUSH DX         ;RETTE DX-REGISTER
;
; STARTE NUN DEN AKTUELLEN CODE FÜR DIE UNTERBRECHUNGSROUTINE
;
                MOV AX,NUM
```

Abb. 6.6: Ein typischer Start einer Unterbrechungs-Serviceroutine, der zeigt, wie wichtige Register auf den Stapel gerettet werden

```
;
; START DER UNTERBRECHUNGSSERVICEROUTINE
;
                STI             ;SCHALTE UNTERBRECHUNGEN WIEDER EIN
                PUSH AX         ;RETTE AX-REGISTER
                PUSH BX         ;RETTE BX-REGISTER
                PUSH CX         ;RETTE CX-REGISTER
                PUSH DX         ;RETTE DX-REGISTER
;
; STARTE NUN DEN AKTUELLEN CODE FÜR DIE UNTERBRECHUNGSROUTINE
;
                MOV AX,NUM
```

Abb. 6.7: Dieses Diagramm zeigt den Start einer Unterbrechungs-Serviceroutine, die weitere Unterbrechungen erlaubt. Dies wird am Anfang der Serviceroutine durch den STI-Befehl ermöglicht

benutzen den IRET-Befehl. Wenn dieser Befehl ausgeführt wird, werden die Kennzeichenbits, das IP und das CS-Register vom Stapel geholt. Vor der Rückkehr zum Hauptprogramm ist es wichtig sicherzustellen, daß die Unterbrechungen wieder freigegeben werden (falls dies gewünscht wird). Weiter ist es notwendig, alle internen Register, die während der Unterbrechungsserviceroutine auf den Systemstapel gebracht wurden, herunterzuholen. Abbildung 6.8 zeigt eine Unterbrechungsserviceroutine, die alle diese Punkte aufzeigt.

```
;
; START DER UNTERBRECHUNGSSERVICEROUTINE
;
                PUSH AX         ;RETTE AX-REGISTER
                PUSH BX         ;RETTE BX-REGISTER
                PUSH CX         ;RETTE CX-REGISTER
                PUSH DX         ;RETTE DX-REGISTER
;
; STARTE NUN DEN AKTUELLEN CODE FÜR DIE UNTERBRECHUNGSROUTINE
;
                MOV AX,NUM
                MUL AX,BX
;
; WEITERE ANWEISUNGEN DER ROUTINE FOLGEN HIER
;
; ENDE DER ROUTINE
;
                POP DX
                POP CX
                POP BX
                POP AX
                STI             ;SCHALTE UNTERBRECHUNGEN EIN
                IRET            ;RÜCKSPRUNG VON UNTERBRECHUNGEN
```

Abb. 6.8: Ein Beispiel für eine Unterbrechungsroutine für den 8086/8088

DAS ZURÜCKSETZEN (RESET) DES 8086/8088

In den vorangegangenen Beispielen haben wir gezeigt, wie die NMI, INTR und SOFTWARE-Unterbrechungen für die CPU arbeiten. Lassen Sie uns nun einen anderen Typ von Unterbrechung behandeln: die RESET-Eingabe auf dem 8086/8088. Wenn die RESET- oder Rücksetzeingabeleitung aktiviert wird, beginnt die CPU ihre Arbeit auf genau bestimmte Weise. Dies erlaubt einen geordneten Start, Neustart oder ein geordnetes Einschalten des Systems.

Auf die Anforderung der RESET-Eingabeleitung werden die CPU-Register in der folgenden Weise initialisiert:

Kennzeichenbits	= Löschen
Befehlszeiger	= 0000H
CS-Register	= FFFFH
DS-Register	= 0000H
SS-Register	= 0000H
ES-Register	= 0000H

Basierend auf diesen Registerdefinitionen wird der erste Befehl nach dem Zurücksetzen aus der folgenden Speicherzelle geholt:

$$(CS \times 16) + IP = FFFF0H + 0000H = FFFF0H$$

Die erste Adresse nach dem Zurücksetzen ist die absolute Speicheradresse FFFF0H. Rufen Sie sich ins Gedächtnis zurück, daß der Hersteller empfiehlt, diese hohen Speicheradressen FFFF0H bis FFFFFH nur für ihren vorgesehenen Zweck zu verwenden. Die Gesamtheit der reservierten Speicheradressen für den 8086/8088 wird in Abbildung 6.9 gezeigt.

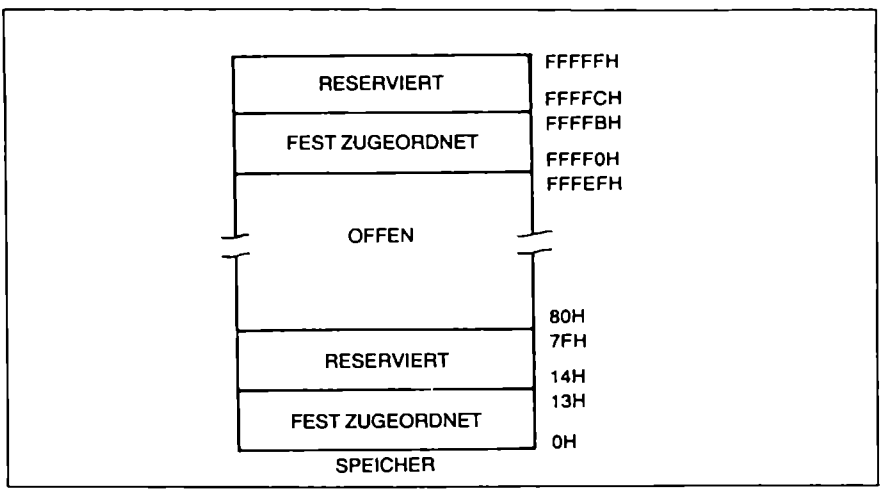


Abb. 6.9: Eine Übersicht über die reservierten Speicherbereiche für den 8086/8088

ZUSAMMENFASSUNG

In diesem Kapitel haben wir Unterbrechungen für den 8086/8088 untersucht. Unsere Erörterung begann mit einer allgemeinen Einführung einer Unterbrechung. Von dort fuhren wir mit der Behandlung beider, der maskierbaren „INTR"- und der nichtmaskierbaren „NMI"-Unterbrechung fort. In jedem Fall haben wir erklärt, wie die Unterbrechungsanforderung von der CPU verarbeitet wird.

Als nächstes haben wir erörtert, wie die CPU interne Unterbrechungen unter Verwendung der Befehle INT und INTO erzeugt. Wir haben das Kapitel abgeschlossen, indem wir gezeigt haben, wie der 8086/8088 intern auf ein Zurücksetzen über die RESET-Leitung antwortet.

Wenn Sie dieses Kapitel einmal verstanden haben, wissen Sie, wie die CPU programmiert werden kann, um verschiedene Unterbrechungsanforderungen von der peripheren Systemhardware zu verarbeiten.

Kapitel 7

Eingabe und Ausgabe für den 8086/8088

Ein 8086/8088-System enthält im allgemeinen eine oder mehrere Eingabe- und/oder Ausgabeeinrichtungen. In diesem Kapitel erörtern wir die Eingabe-/Ausgabe-Architektur der 8086/8088-CPU. Wir stellen auch die speziellen Befehle zur E/A vor und erklären, wie sie benutzt werden können. Wir geben mehrere Beispiele an, die zeigen, wie ein 8086/8088 mit den verschiedenen E/A-Einrichtungen kommuniziert und sie kontrolliert.

WAS IST EINGABE UND AUSGABE?

In Kapitel 2 haben wir den allgemeinen Aufbau eines auf einem Mikroprozessor basierenden Systems untersucht. Abbildung 7.1 gibt einen kurzen Rückblick. In dieser Abbildung können wir erkennen, daß der Mikroprozessor mit ROM, RAM und E/A kommuniziert. ROM und RAM sind zusammengefaßt und machen den Systemspeicher aus (Eingabe/Ausgabe). Beim 8086/8088 hat der Systemspeicher Adressen von 00000H bis FFFFFH. Beachten Sie, daß der E/A-Block nicht in diesem Speicherraum enthalten ist.

Einige Mikroprozessoren weisen einen Teilbereich des verfügbaren Systemspeichers der E/A zu. Diese Gruppe von Mikroprozessoren umfaßt den 6800, den 6502, den 6809 und den 68 000 – um nur einige zu nennen. Von Mikroprozessoren, die zur E/A Speicherplatz verwenden, sagt man, daß sie „memory mapped I/O (input/output)" verwenden. Das bedeutet, daß die E/A-Geräte wie Speicherzellen adressiert werden.

Der 8086 und der 8088 verwenden kein „memory mapped I/O". So kann der gesamte Systemspeicherplatz als Speicher benutzt werden, weil die System-E/A ihren eigenen Adreßbereich hat. Eine E/A-Architektur wie diese heißt „I/O mapped I/O".

Eine E/A-Operation kann wie folgt definiert werden:

> ***Eingabe:*** Wenn der Mikroprozessor Daten aus einer Quelle liest, die nicht der Systemspeicher ist.

Ausgabe: Wenn der Mikroprozessor Daten in ein Ziel schreibt, das nicht der Systemspeicher ist.

Früher in diesem Text haben wir festgestellt, daß der Systemkontrollbus den Typ der ablaufenden Kommunikation definiert. Mit anderen Worten: Wenn das System „I/O mapped I/O" benutzt, gibt es getrennte Kontrolleitungen für die E/A- und die Speichersysteme. Das Speichersystem verwendet zum Beispiel Kontrolleitungen, die mit Speicherlese- und Speicherschreibleitung bezeichnet sind, während das E/A-System Kontrolleitungen verwendet, die mit Eingabelese- und Ausgabeschreibleitung bezeichnet werden. Siehe Abbildung 7.2.

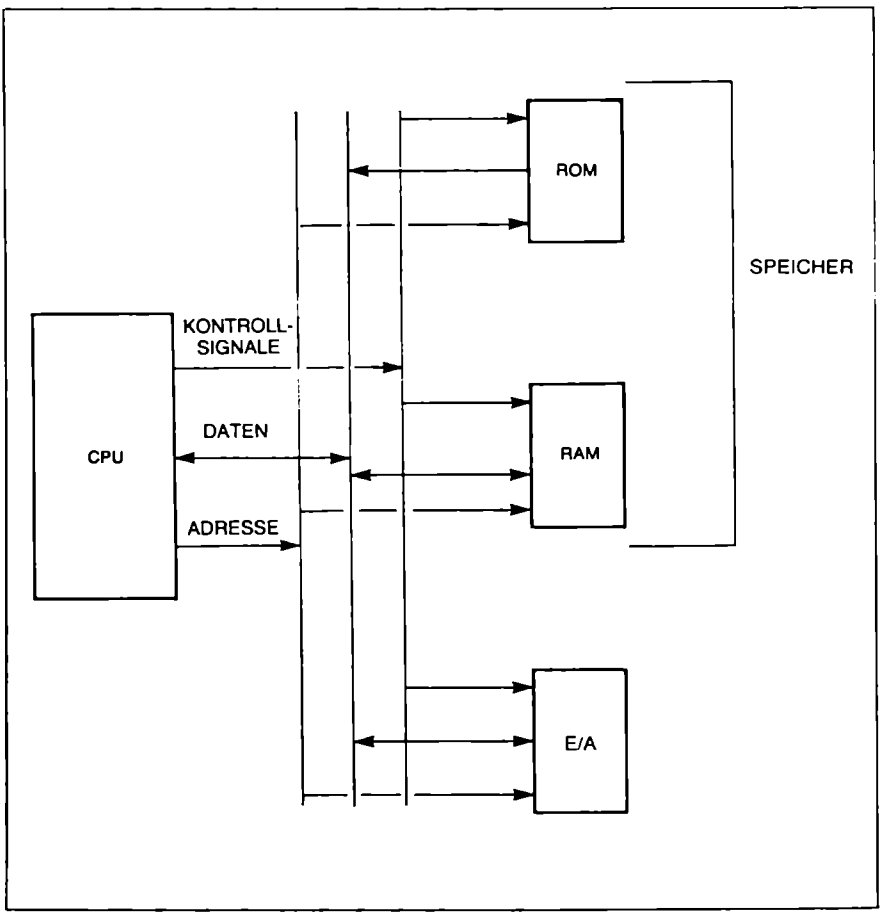


Abb. 7.1: Dieses Diagramm zeigt den allgemeinen 3-Bus-Aufbau eines Mikroprozessorsystems

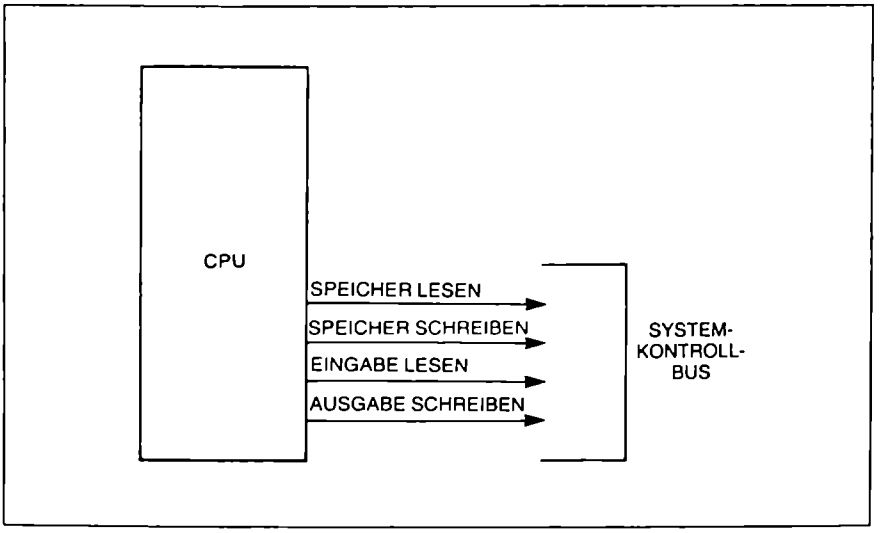


Abb. 7.2: Die Systemkontrollsignale für den Speicher sind „Speicher lesen" und „Speicher schreiben". Die entsprechenden Kontrollsignale für E/A sind „Eingabe lesen" und „Eingabe schreiben"

E/A-ADRESSIERUNG

In einem typischen Mikroprozessorsystem gibt es gewöhnlich mehrere E/A-Anschlüsse (I/O ports). (Ein *Anschluß* (port) ist eine eindeutige Stelle zum Lesen oder Schreiben von Daten, die nicht im Systemspeicher sind. In der Tat ist ein Anschluß einer Stelle im Systemspeicher ähnlich, aus der oder in die Daten während einer Speicheroperation gelesen oder geschrieben werden.) Jedem Anschluß im E/A-System ist eine bestimmte Adresse, *Anschlußauswahlcode* genannt, zugeordnet (siehe Abbildung 7.3). Zur Erzeugung des Anschlußauswahlcodes werden die Systemadreßleitungen A0-A15 mit einem Logikteil entschlüsselt, um auf eine bestimmte Kombination zu antworten. Ein Anschluß kann zum Beispiel einen Anschlußauswahlcode von 0057H haben, wie in Abbildung 7.4 gezeigt wird.

Dem 8086/8088 stehen nur 16 Adreßleitungen der (insgesamt) 20 Mikroprozessoradreßleitungen für die E/A-Adressierung zur Verfügung. Dies ergibt eine Summe von 65536 Ein- und Ausgabeanschlüssen, die in dem System zur Verfügung stehen. Abbildung 7.5 zeigt ein Speicherabbild des Bereichs, der für die E/A mit dem 8086/8088 zur Verfügung steht.
Obwohl das E/A-System und das Speichersystem völlig getrennt sind, benutzen diese beiden Systeme dieselben Adreßleitungen. Dies bedeutet, daß Sie z.B. eine Speicheradresse 00F4H und eine E/A-Adresse

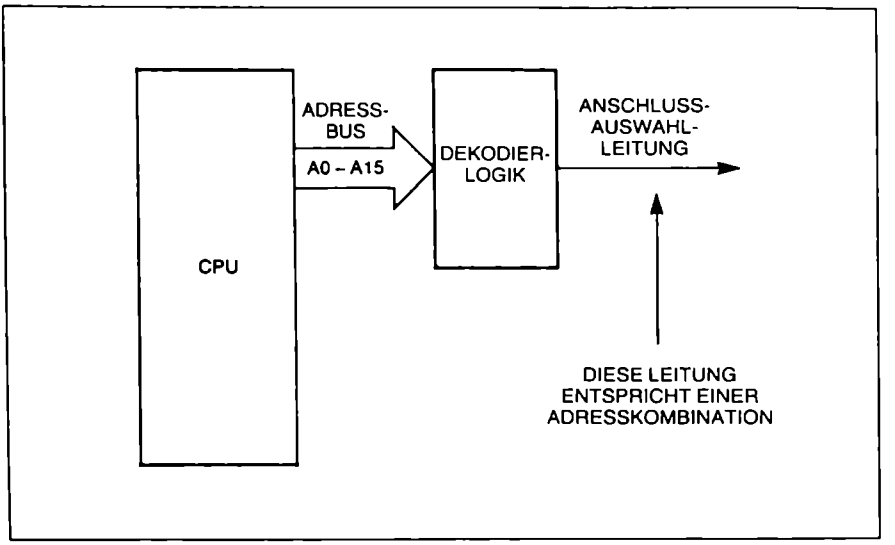


Abb. 7.3: Jeder E/A-Anschluß ist so gestaltet, daß er auf eine eindeutige Adreßkombination anspricht

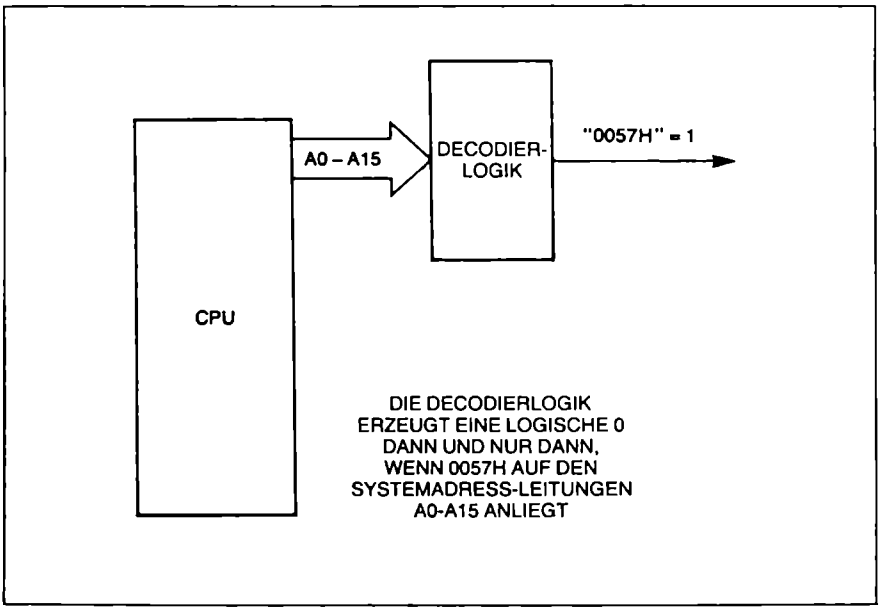


Abb. 7.4: Dieser E/A-Anschluß spricht an, wenn die Systemadresse 0057H am Adreßbus anliegt

00F4H haben können. Diese beiden Systeme können durch einen Blick auf die Leitungen im Systemkontrollbus unterschieden werden: Während einer Speicheroperation ist die Speicherlese- oder Speicherschreibkontrolleitung aktiv, und während einer E/A-Operation ist die Eingabelese- oder die Ausgabeschreibleitung aktiv.

Reservierter E/A-Raum

Abbildung 7.5 zeigt einen Bereich des E/A-Raumes – die Adressen 00F8H-00FFH – der als „reserviert" markiert ist. Dieser Bereich ist deshalb als „reserviert" markiert, weil die Firma Intel vorgesehen hat, daß er nicht in Ihrer Systemanwendung, aber bei künftigen Intelprodukten benutzt wird.

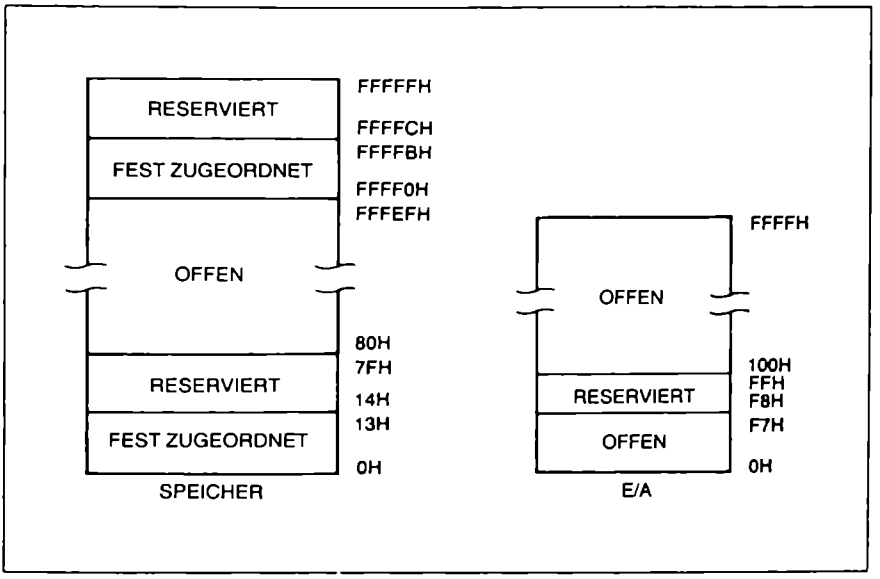


Abb. 7.5: Eine Übersicht über den verfügbaren Bereich für Systemspeicher und E/A

Was ist eine E/A-Einrichtung?

Eine *E/A-Einrichtung* kann als Hardware definiert werden, die vom System kontrolliert wird. So eine Einrichtung kann eine oder mehrere E/A-Anschlüsse oder E/A-Adressen haben, die dazu gehören, wie in Abbildung 7.6 gezeigt wird. Beispiele für E/A-Einrichtungen sind LSI-Bausteine (LSI: large scale integration oder Großintegration), wie z.B. ein Diskettensteuerbaustein oder ein Zeitgeberbaustein.

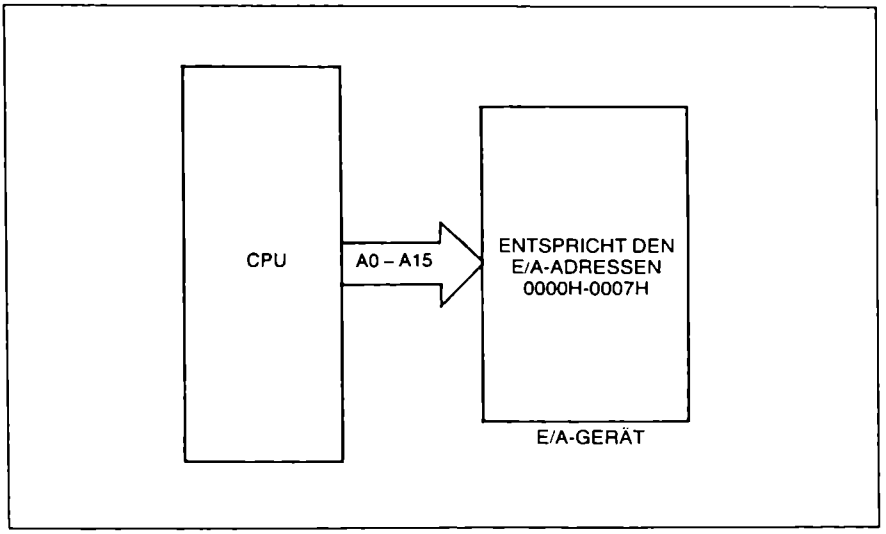


Abb. 7.6: Ein E/A-Gerät spricht auf eine oder mehrere E/A-Anschlußadressen an. Dieses Gerät spricht auf alle E/A-Adressen zwischen 0000H und 0007H einschließlich an.

EINGABEBEFEHL

Der IN-Befehl wird benutzt, um Daten von einem Eingabeanschluß in den Mikroprozessor zu lesen. Dieser Befehl wird wie folgt geschrieben:

IN Akkumulator, Anschluß

Der IN-Befehl überträgt ein Datenbyte oder Datenwort von einer Eingabeanschlußadresse in den Akkumulator. Wenn ein Datenbyte übertragen wird, wird es im AL-Register gespeichert; wenn es sich um ein Wort handelt, so wird es in das AX-Register geschrieben.

Adressierung eines festen Anschlusses

Es gibt zwei Arten, den IN-Befehl zu benutzen – entweder mit fester oder mit variabler Anschlußadresse. Wenn ein IN-Befehl mit *fester Anschlußadresse* benutzt wird, wird er wie in Abbildung 7.7 gezeigt verschlüsselt. In dieser Abbildung können wir erkennen, daß das W-Bit im ersten Byte logisch 1 oder 0 gesetzt werden kann. Wenn das W-Bit logisch 1 ist, wird ein Wort (16 Bits) übertragen und die Daten werden schließlich im AX-Register gespeichert. Wenn es logisch 0 ist, wird ein Byte übertragen und die Daten werden im AL-Register gespeichert.

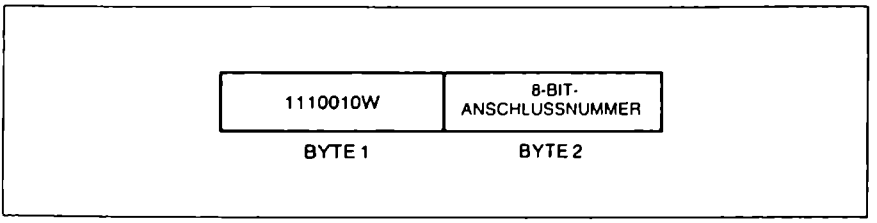


Abb. 7.7: Die Codierung für einen Eingabebefehl (IN) mit fester Anschlußadresse

Das zweite in Abbildung 7.7 gezeigte Byte ist eine 8-Bit Anschlußadresse. Da es nur 8 Bits gibt, können mit diesem Befehl insgesamt 256 einzelne E/A-Anschlüsse angesprochen werden. Als Beispiel für die Anwendung dieses Befehls lassen Sie uns annehmen, daß wir ein Datenbyte vom Anschluß 3DH lesen. Die Befehlsverschlüsselung sollte wie in Abbildung 7.8 aussehen. Die Art zu spezifizieren, ob es sich um eine Byte oder Wortübertragung handelt, ist wie folgt:

```
IN  AX,  Anschluß  (Wort)
IN  AL,  Anschluß  (Byte)
```

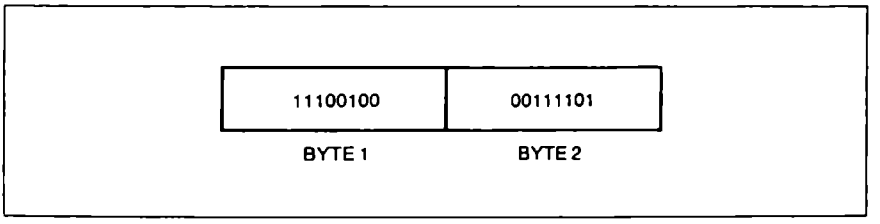


Abb. 7.8: Die Codierung für ein Beispiel eines Eingabebefehls (IN) mit fester Anschlußadresse: IN AL,3DH

Adressierung eines Anschlusses mit variabler Adresse

Eine andere Art den IN-Befehl zu benutzen ist die mit variabler Anschlußadresse. Wenn Sie die Codierung für eine variable Anschlußadresse wählen, haben Sie Zugang zu allen 16 Bits des Systemadreßbusses für die E/A-Adressierung. Die Verschlüsselung des IN-Befehlsformats mit variablem Anschluß zeigt Abbildung 7.9.

Bei der Betrachtung von Abbildung 7.9 können wir erkennen, daß zur Benutzung dieses Befehls nur ein Byte erforderlich ist. Das W-Bit ist gleich 1, wenn ein Datenwort gewünscht wird; es ist gleich 0, wenn ein Datenbyte gewünscht wird. Die Adresse des Anschlusses ist zur Zeit der Befehlsausführung im DX-Register enthalten. Durch die Benutzung des

DX-Registers stehen 16 Adreßbits zur Verfügung, die unter Steuerung des Programms modifiziert werden können. Zum Beispiel können wir das IN-Befehlsformat für einen variablen Anschluß benutzen, um ein Datenwort von Anschluß 05ACH zu lesen. Das in Abbildung 7.10 gezeigte Programm leistet dies.

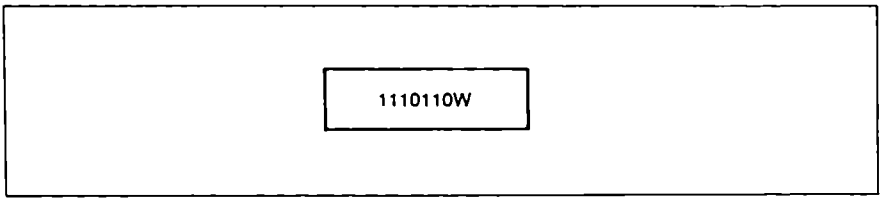


Abb. 7.9: So sieht die Codierung für einen Eingabebefehl (IN) mit variabler Anschlußadresse aus. Die Adresse ist im DX-Register der CPU enthalten

```
                    1 ;
                    2 ; EIN PROGRAMM ZUR BENUTZUNG DES MODUS MIT VARIABLER
                    3 ; ANSCHLUSSADRESSE FÜR DEN IN-BEFEHL
                    4 ; 16-BIT-ANSCHLUSSNUMMER = 05AC
                    5 ;
0000 BAAC05         6 MOV DX,# 05ACH   ;ANSCHLUSSNUMMER IN DAS DX-REGISTER
0003 ED             7 IN AX,DX         ;EINGABE EINES WORTES NACH AX ÜBER
                    8                  ;ANSCHLUSSADRESSE IN DX
```

Abb. 7.10: Ein 8086/8088-Programm das die Benutzung des Befehls IN mit variabler Anschlußadresse zeigt

AUSGABEBEFEHL

Der OUT-Befehl wird benutzt, um Daten der CPU zu einem Ausgabeanschluß zu übertragen. Dieser Befehl hat ein ähnliches Format wie der IN-Befehl. Sein Format ist:

OUT Anschluß, Akkumulator

Wie der IN-Befehl kann der Out-Befehl in den Formaten mit fester oder variabler Anschlußadresse verwendet werden. Abbildung 7.11 zeigt Beispiele zweier kleiner Programme, die den OUT-Befehl in den Formaten mit fester und variabler Anschlußadresse verwenden.

```
                    1 ;
                    2 ;
0000 BAAC05         3  MOV DX, = 05ACH    ;LADE ADRESSE IN DAS DX-REG
0003 EC             4  IN AL,DX           ;LIES BYTE NACH AL VON ADDR.
                    5                     ;IN DX
                    6 ; GIB NUN DAS BYTE MIT VARIABLEM ANSCHLUSSFORMAT AUS
                    7 ;
0004 BA8704         8  MOV DX, = 0487H    ;LADE ADRESSE IN DAS DX-REG
0007 EE             9  OUT DX,AL          ;AUSGABE DER DATEN MIT
                   10                     ;VARIABLER ANSCHLUSSADRESSE
                   11 ;
0008 E607          12  OUT 07H,AL         ;AUSGABE DES BYTES ZU FESTER
                   13                     ,ANSCHLUSSADRESSE = 07
                   14 ;
```

Abb. 7.11: Dieses 8086/8088-Programm zeigt, wie der Befehl OUT mit fester und mit variabler Anschlußadresse benutzt wird

8086-EINGABE- UND AUSGABEANSCHLÜSSE

Wenn wir den 8086 benutzen, müssen wir uns daran erinnern, daß er einen 16-Bit-Datenbus hat. Dies bedeutet, daß die Adressen der Anschlüsse mit dem Datenbyte (entweder dem höherwertigen oder dem niederwertigen) übereinstimmen müssen. Wenn ein Byteanschluß eine gerade Adresse hat, muß er physikalisch mit dem niederwertigen Byte (D0-D7) des 16-Bit-Datenbusses verbunden sein. Ein Byteanschluß mit einer ungeraden Adresse muß physikalisch mit dem oberen Byte (D8-D15) des Systemdatenbusses verbunden sein. Eine Wortanschlußadresse sollte als gerade Adresse spezifiziert werden, wie Abbildung 7.12 zeigt.

8088-EINGABE- UND AUSGABEANSCHLÜSSE

Wenn Sie die 8088-CPU verwenden, die einen 8-Bit-Datenbus hat, können Sie sowohl Wort- als auch Byteausgabeanschlüsse benutzen. Dieselben Vereinbarungen, die wir für den 8086 erörtert haben, können für den 8088 befolgt werden. Es ist jedoch nicht notwendig, sich mit einem gerade oder ungerade adressierten Byteanschluß zu beschäftigen, da es nur eine Art der Verbindung mit dem 8-Bit-Datenbus gibt.

Wenn sie einen Wortausgabeanschluß verwenden, sollte das niederwertige Byte des Wortanschlusses an einer geraden Adresse und das höherwertige Byte an einer ungeraden Adresse stehen. Dies stimmt mit der Organisation des 8088-Systemspeichers überein, wie Abbildung 7.13 zeigt.

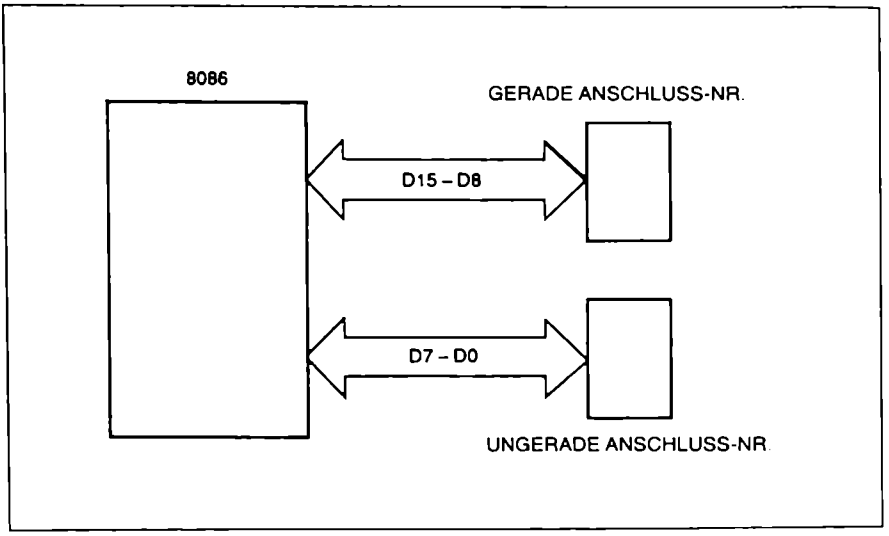


Abb .7.12: Dieses Blockdiagramm zeigt, daß bei einem 8086-System ein 8-Bit-E/A-Anschluß an einer ungeraden Adresse mit den oberen Datenleitungen (D8-D15) verbunden werden muß, und daß ein E/A-Anschluß an einer geraden Adresse mit den unteren Datenleitungen (D0-D7) Verbunden werden muß

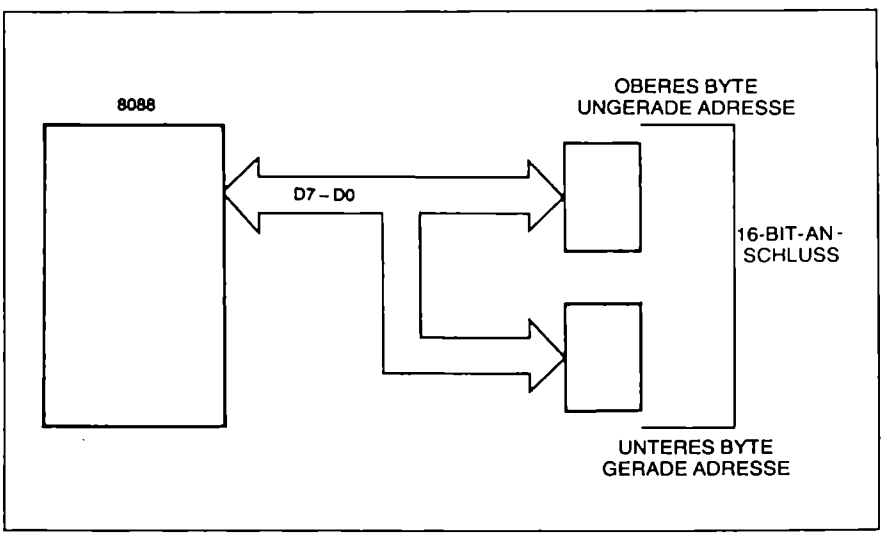


Abb. 7.13: E/A-Anschlüsse in einem 8088-System können gerade oder ungerade Adressen haben. Falls ein 16-Bit-Anschluß festgelegt wird, sollte das untere Byte an einer geraden Adresse und das obere Byte an einer ungeraden Adresse sein

Erzeugung eines Signals mit Hilfe von Software

Lassen Sie uns nun die Kenntnisse, die wir über die 8086/8088- Befehle und E/A erhalten haben, auf verschiedene allgemeine Probleme und Schnittstellen, die in Mikrocomputersystemen benutzt werden, anwenden. Wir beginnen mit der Erzeugung eines Signals, das ein Hardwareelement, wie z.B. ein Relais, ein- oder ausschaltet.

Um ein Signal zu erzeugen, muß der Computer eine Ausgabeeinrichtung ein- oder ausschalten. Um dies zu leisten, muß der Computer ein elektrisches Spannungsniveau in der Einrichtung von logisch 0 zu logisch 1 oder von logisch 1 zu logisch 0 verändern. Lassen Sie uns zum Beispiel annehmen, daß ein externes Relais mit Bit 0 eines 8-Bit-Ausgabeanschlusses, den wir OUT1 nennen, verbunden ist. (OUT1 ist der Variablenname, der die Adresse [8 oder 16 Bits] des Ausgabeanschlusses darstellt.)

Um das Relais einzuschalten, muß eine logische 1 in Bit 0 von OUT1 geschrieben werden. Um das Relais auszuschalten, muß eine logische 0 dorthin geschrieben werden. Hier ist ein Programm, das das Relais einschaltet:

```
MOV  AL,00000001B  ;B = BINÄR
OUT  OUT1,AL       ;GIB 1 AUF BIT 0 VON ANSCHLUSS
                   ;OUT1 AUS
```

In diesem Beispiel haben wir angenommen, daß die Zustände der anderen Bitpositionen des Ausgabeanschlusses „uninteressant" sind. In einer normalen Situation ist dies jedoch möglicherweise nicht der Fall; sie können mit anderen Relais des Systems verbunden sein. In diesem Fall wollten wir den logischen Zustand dieser Bits nicht verändern. Wir können sicherstellen, daß dies nicht geschieht, indem wir das Programm folgendermaßen verändern:

```
IN   AL,OUT1       ;LIES DEN WERT DES
                   ;AUSGABEANSCHLUSSES
OR   AL,00000001B  ;SETZE NUR BIT 0, ALLE ANDEREN
                   ;BITS BLEIBEN UNVERÄNDERT
OUT  OUT1,AL       ;GIB DEN INHALT DES AL-REGI-
                   ;STERS AN
                   ;ANSCHLUSS OUT1 AUS
```

Dieses Programm nimmt an, daß wir den Inhalt des Ausgabeanschlusses OUT1 lesen können, oder daß der Inhalt in eine Speicherzelle, auf die wir zugreifen können, gerettet worden ist. Abbildung 7.14 zeigt ein Blockdiagramm des Konzepts, ein Relais ein- und auszuschalten.

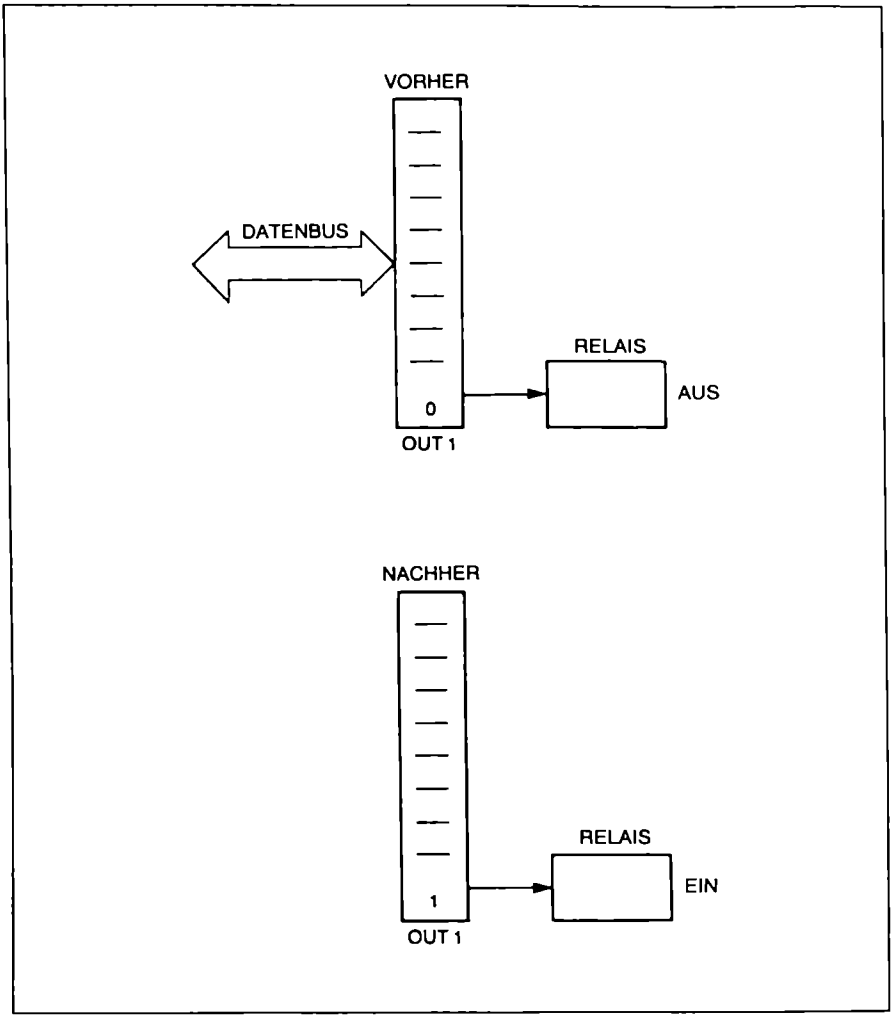


Abb. 7.14: Dieses Blockdiagramm zeigt, wie eine Signalleitung eines Ausgabeanschlusses zum Ein- und Ausschalten eines Relais benutzt werden kann

Erzeugung eines Impulses

Wir können einen Impuls auf die gleiche Art erzeugen, auf die wir ein Relais ein- und ausschalten. Zuerst setzen wir ein Bit in einem Ausgabeanschluß auf logisch 1 und setzen es dann eine bestimmte Zeit später wieder auf logisch 0 (siehe Abbildung 7.15). Wenn der Impuls erzeugt ist, müssen wir seine Zeitdauer mit Hilfe von Software steuern. Um dies zu tun, müssen wir eine berechnete Verzögerung erzeugen.

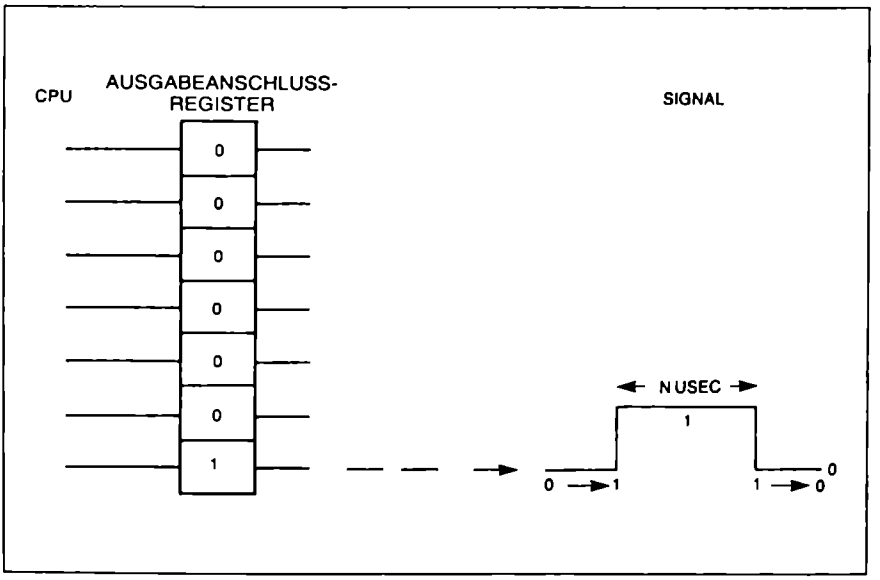


Abb. 7.15: Man kann einen Impuls erzeugen, indem man ein Bit eines Ausgabeanschlusses für eine bestimmte Dauer hochsetzt und dann wieder auf logisch 0 zurücksetzt

Erzeugung einer Verzögerung

Verzögerungen können mit Hilfe von Software oder Hardware erzeugt werden. Jetzt wollen wir Software benutzen. In einem späteren Beispiel werden wir eine Hardwareverzögerung mit Hilfe eines programmierbaren Zeitgebers erzeugen.

Programmierte Verzögerungen werden durch Zählen erreicht. Um zu zählen, wird zuerst ein Wert in ein Zählregister geladen und dann vermindert. Das Programm enthält eine Schleife und vermindert solange bis der Zähler den Wert 0 erreicht. Die Gesamtdauer dieses Prozesses entspricht der gewünschten Verzögerung.

Um eine bestimmte Verzögerung zu erzeugen, müssen wir wissen, wieviel Ausführungszeit jeder Befehl bei einem vorgebenen Zeittakt benötigt. Ein Befehl benötigt zur Ausführung eine feste Anzahl von Zeittakten und diese Anzahl muß in Zeit umgewandelt werden, die auf der Periode der Systemuhr basiert. (In diesem Beispiel beschäftigen wir uns nicht mit exakten Zeiten.)

Nun wollen wir eine Verzögerung der Dauer erzeugen, die benötigt wird, um das AX-Register in einer Schleife zu vermindern. Hier ist das Programm:

```
        MOV   AX, 0F45H ;LADE AX
                        ;MIT DEM WERT 0F45H
BACK    DEC   AX        ;AX = AX - 1
        JNZ   BACK      ;WENN AX UNGLEICH NULL IST,
                        ;VERMINDERE WEITER
```

Die erste Befehlszeile lädt das AX-Register mit dem Direktoperanden F45H. Die nächste mit „BACK" markierte Zeile vermindert das AX-Register um eins. Anschließend bewirkt der JNZ-Befehl (jump if not equal to zero, springe falls ungleich Null) einen Sprung in die mit „BACK" markierte Befehlszeile, wenn das Ergebnis der Verminderung ungleich Null ist. Wenn der Wert des AX-Registers gleich Null ist, führt das Programm den dem JNZ-Befehl folgenden Befehl aus. Ein Flußdiagramm, das diese Art der Verzögerung zeigt, erscheint in Abbildung 7.16.

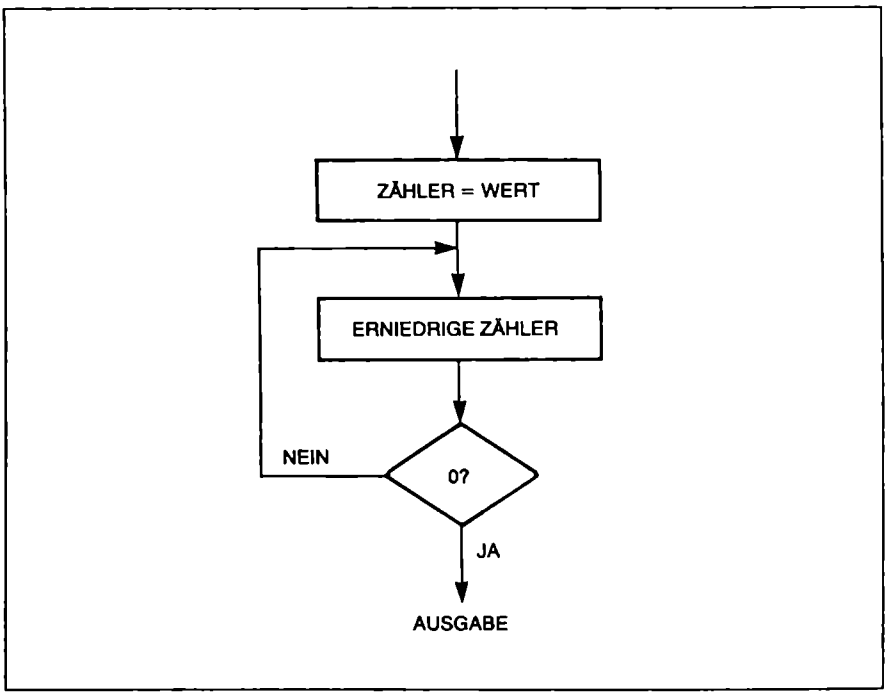


Abb. 7.16: Dieses Flußdiagramm zeigt, wie eine Verzögerung durch Software erzeugt werden kann

Abbildung 7.17 stellt ein Programm zur Erzeugung eines Impulses an Bit 0 des Ausgabeanschlusses OUT1 dar. Beachten Sie, daß dieses Programm das soeben beschriebene Verzögerungsprogramm benutzt.

```
                    1 ;
                    2 ; EIN PROGRAMM ZUR ERZEUGUNG EINER VERZÖGERUNG
                    3 ; AX WIRD BENUTZT
                    4 ;
                    5 ; AUSGABEANSCHLUSS OUT1 = 58H
                    6 ; BIT 0 IST DAS KONTROLLBIT
                    7 ;
                    8 OUT1      EQU 58H
                    9 ;
0000 B001           10          MOV AL,# 01     ;SETZE DAS KONTROLLBIT
0002 E658           11          OUT OUT1,AL     ;GIB DAS KONTROLLBIT AUS
0004 BB5A04         12          MOV BX,#045AH;LADE BX MIT ZÄHLERWERT
0007 4B             13 BACK     DEC BX          ;BX=BX-1
0008 75FD           14          JNZ BACK        ;BX≠0, DEKREMENTIERE ERNEUT
000A FEC8           15          DEC AL          ;AL=0
000C E658           16          OUT OUT1,AL     ;KONTROLLBIT=0
                    17 ;
                    18 ; DAS AUSGABESIGNAL IST AUF LOGISCH 0 ZURÜCKGESETZT
```

Abb. 7.17: Dies ist ein 8086/8088-Programm zur Erzeugung eines Impulses unter Verwendung der Technik der Software-Verzögerung, die in Abb. 7.16 vorgestellt wurde

Längere Verzögerungen

Mit dem 8086/8088 können wir leicht kurze Verzögerungen erzeugen, indem wir entweder 8-Bit- oder 16-Bit-Register benutzen. Wenn wir jedoch eine längere Verzögerungszeit benötigen, können wir zwei 16-Bit-Register und geschachtelte Schleifen verwenden (wie das Flußdiagramm in Abbildung 7.18 zeigt). Ein Programm zur Realisierung dieses Flußdiagramms wird in Abbildung 7.19 angegeben.

DIE BENUTZUNG EINES 8255-PIO-BAUSTEINS MIT DEM 8086/8088

Wir wollen nun zeigen, wie der 8086/8088 zur Steuerung des 8255, einer programmierbaren Ein-/Ausgabeeinrichtung, benutzt werden kann. Diese Bausteine werden häufig in einem Mikroprozessorsystem verwendet, um der CPU zu erlauben, periphere Hardware zu steuern. Wir beginnen mit einem kurzen Überblick über den 8255; dann fahren wir fort mit der Untersuchung einiger in 8086/8088-Assemblersprache geschriebener Softwareroutinen zur Steuerung des Bausteins.

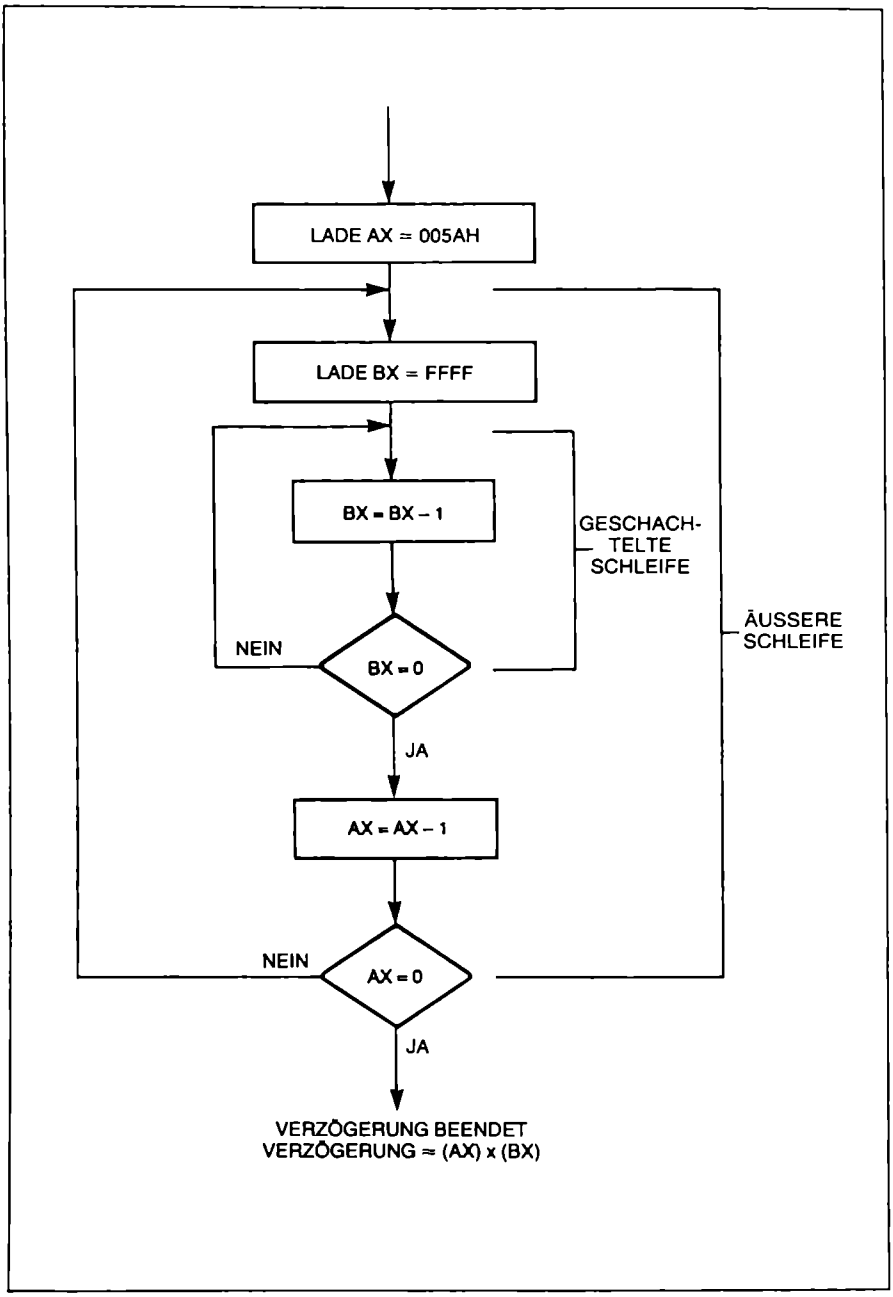


Abb. 7.18: Dieses Flußdiagramm zeigt, wie eine längere Verzögerung unter Benutzung zweier interner CPU-Register erzeugt werden kann

```
                    1 ;
                    2 ; EIN PROGRAMM ZUR ERZEUGUNG EINER VERZÖGERUNG, DAS
                    3 ; DIE REGISTER AX UND BX BENUTZT.
                    4 ; AX IST DER ÄUSSERE, BX DER INNERE SCHLEIFENZÄHLER
                    5 ;
0000 B85A00         6           MOV AX,#0005AH    ;LADE DAS REGISTER AX
0003 BBFFFF         7 BACK      MOV BX,#0FFFFH    ;LADE DAS REGISTER BX
0006 4B             8 BACK1     DEC BX            ;BX=BX-1
0007 75FD           9           JNZ BACK1         ;INNERE SCHLEIFE≠0
0009 48            10           DEC AX            ;AX=AX-1
000A 75F7          11           JNZ BACK          ;ÄUSSERE SCHLEIFE≠0
                   12 ;
                   13 ; DIE VERZÖGERUNG IST ANNÄHERND GLEICH AX MAL BX
                   14 ;
```

Abb. 7.19: Dies ist ein 8086/8088-Programm zur Realisierung der Verzögerungsschleife aus Abb. 7.18.

Überblick über den 8255

Der 8255 ist ein LSI-Baustein mit einem Standard-IC-Gehäuse mit 40 Anschlußstiften im Abstand von 0.1 Zoll, die in zwei Reihen parallel angeordnet sind. Dieser Baustein ist zur Ausführung einer Vielfalt von Schnittstellenaufgaben in einer Mikroprozessorsystemumgebung konstruiert. Abbildung 7.20 zeigt ein Blockdiagramm des 8255-Bausteins. Lassen Sie uns dieses Diagramm untersuchen und im allgemeinen die Funktion jedes einzelnen Blocks besprechen.

In diesem Diagramm gibt es vier Blöcke, die den 8255 physikalisch mit externer Hardware verbinden. Diese Blöcke haben Steuerleitungen, die mit PA0-PA7, PB0-PB7, PC0-PC3, PC4-PC7 bezeichnet sind. Die Gruppen von Signalen aus diesen Blöcken werden logisch in drei verschiedene E/A-Anschlüsse unterteilt, die mit PORT A (PA), PORT B (PB) und PORT C (PC) bezeichnet sind. Diese vier Anschlußblöcke (PC ist in zwei Gruppen unterteilt) sind mit einem internen Datenbus auf dem 8255-Baustein verbunden, mit dessen Hilfe die Anschlüsse programmiert werden.

In Abbildung 7.20 gibt es zwei Blöcke. Diese Blöcke, die mit Steuerung Gruppe A und Steuerung Gruppe B benannt sind, definieren, wie die drei E/A-Anschlüsse in dem System arbeiten sollen. Für den 8255 gibt es mehrere verschiedene Arbeitsweisen, und diese müssen von der CPU durch das Schreiben von Steuerworten auf den Baustein definiert werden. Beachten Sie, daß Gruppe C des 8255 aus zwei 4-Bit-Anschlüssen besteht. Eine der 4-Bit-Gruppen ist mit dem Steuerungssignal der Gruppe A und die andere mit dem Steuerungssignal der Gruppe B verbunden.

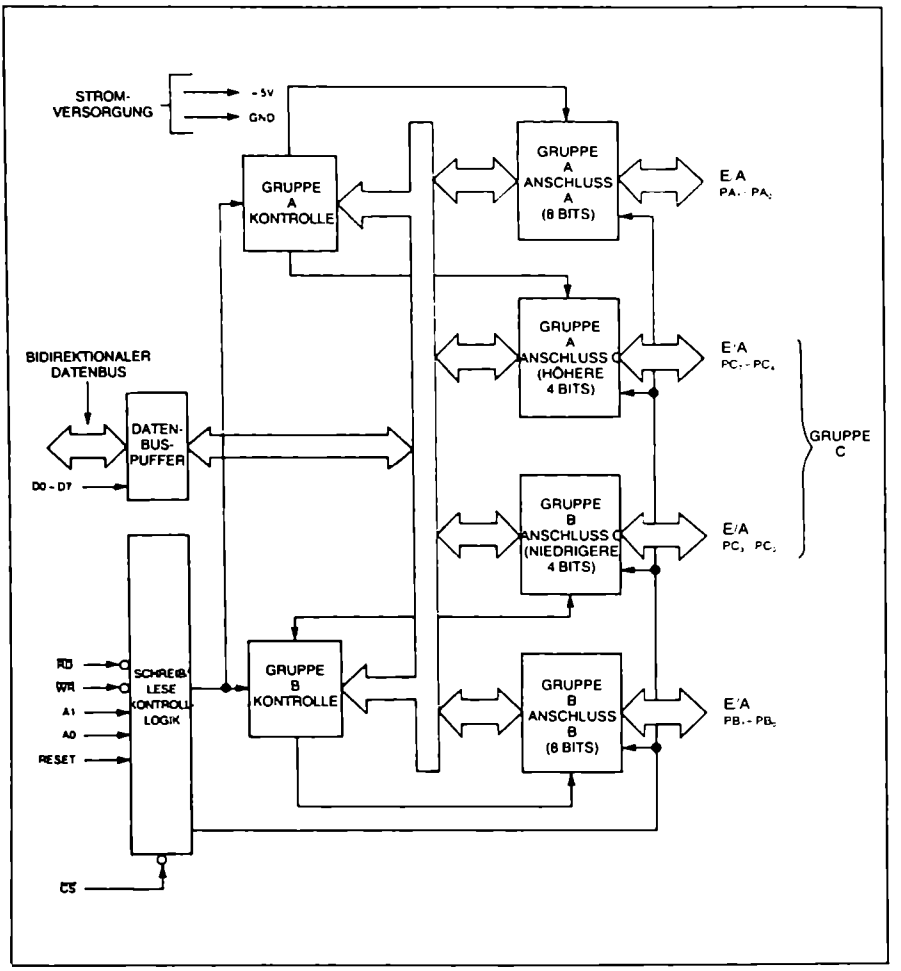


Abb. 7.20: Ein Blockdiagramm der 8255-PIO

Die verbleibenden logischen Blöcke in Abbildung 7.20 sind als Datenbuspuffer und Lese-/Schreibsteuerlogik bezeichnet. Diese beiden Blöcke bilden die elektrische Schnittstelle zwischen dem 8086/8088-Mikroprozessor und dem 8255. Der Datenbuspuffer puffert die Dateneingabe- und Datenausgabeleitungen zum und vom CPU-Datenbus. Die Lese-/Schreibsteuerlogik transportiert die Daten zu und von den richtigen internen Registern mit der passenden Synchronisierung. Diese Synchronisierung hängt von der Art der Operation ab, die von der CPU ausgeführt wird; das heißt, sie hängt davon ab, ob es sich um eine E/A-Lese- oder Schreiboperation handelt oder nicht.

Physikalische Verbindung mit dem CPU-System

Abbildung 7.21 stellt ein Blockdiagramm vor, das zeigt, wie der 8255 mit dem 8086/8088-System verbunden ist. Beachten Sie, daß es vier interne Anschlüsse gibt, die durch die Adreßleitungen A1 und A0 angesprochen werden. Wir werden die Anschlußadressen 10H, 11H, 12H und 13H für den 8088, und 10H, 12H, 14H und 16H für den 8086 benutzen. Der Unterschied zwischen diesen beiden Mengen von Adressen liegt in der Tatsache, daß das System die Daten auf den oberen Datenleitungen (D8-D15) überträgt, wenn vom 8086 eine ungerade Adresse geschrieben wird. Deshalb müssen alle Anschlußadressen gerade sein, wenn wir die unteren Datenleitungen (D0-D7) des 8086 anschließen.

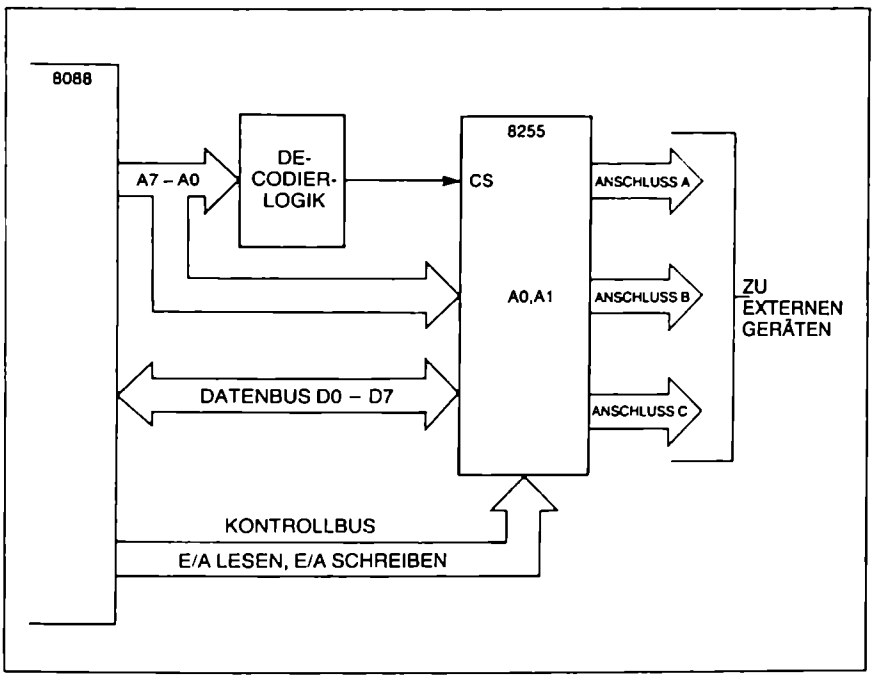


Abb. 7.21: Dieses Diagramm zeigt die Verbindung des 8255 mit der 8088-CPU. Falls der 8255 mit dem 8086 verbunden werden soll, müssen Sie entscheiden, welches Byte des 16-Bit-Datenbusses angeschlossen werden soll

8255-Lese- und Schreibregister

Lassen Sie uns nun die Registerdefinitionen und Anschlußadressen für das in Abbildung 7.21 gezeigte 8255-System untersuchen. Wir wollen annehmen, daß eine 8088-CPU vorliegt. Hier sind die Registerdefinitionen:

Anschluß-nummer	Definition	Beschreibung
10H	Schreibe ANSCHLUSS-A-Daten	Ausgabe
10H	Lies ANSCHLUSS-A-Daten	Eingabe
11H	Schreibe ANSCHLUSS-B-Daten	Ausgabe
11H	Lies ANSCHLUSS-B-Daten	Eingabe
12H	Schreibe ANSCHLUSS-C-Daten	Ausgabe
12H	Lies ANSCHLUSS-C-Daten	Eingabe
13H	Schreibe Steuerwort	Ausgabe
13H	Unzulässiges Leseregister	

Die Funktion der Register 0–2 wird durch das Wort definiert, das in das Kontrollregister 3 des 8255 geschrieben wird. Abbildung 7.22 zeigt die Bit-Definitionen des Kontrollregisters. Wir werden auf diese Definitionen verweisen, da wir ein Programm zur Benutzung des 8255 schreiben.

Basis-Ein- und Ausgabe mit dem 8255

Um den 8255 in dem Basis-E/A-Modus für E/A zu benutzen, muß der Programmierer zuerst ein Byte in das Kontrollregister schreiben. Dieses Byte definiert, wie die Register des 8255-Bausteins zu benutzen sind. Die Bits des Kontrollregisters, die zur Programmierung der grundlegenden E/A-Funktionen benutzt werden, sehen so aus:

1	0	0	0	0	0	0	0
D7	D6	D5	D4	D3	D2	D1	D0

Auf Abbildung 7.22 erkennen wir, daß Bit 7 dieses Byte als ein Kontrollwort definiert, das die Art der Operation festlegt. Lassen Sie uns jedes einzelne Bit untersuchen:

Die Bits D6 und D5 definieren den Modus der Operation für den Anschluß A des 8255. Das ist Modus 0.

Bit D4 = 0 zeigt an, daß Anschluß A ein Ausgabeanschluß ist.

Bit D3 = 0 legt die oberen 4 Bits von Anschluß C als Ausgabe fest.

Bit D2 stellt den Modus für die Anschlußgruppe B ein. Eine logische 0 definiert B als Ausgabeanschluß.

Bit D1 = 0 legt Anschluß B als Ausgabeanschluß fest.

Bit D0 = 0 legt die unteren vier Bits von Anschluß C als Ausgabeanschluß fest.

Mit diesem zum 8255 geschriebenen Kontrollwort sind alle drei Anschlüsse (A, B und C) als Ausgabeanschlüsse definiert. Dies führt zu vierundzwanzig separaten Ausgabeleitungen des 8255, die zur Verbindung mit externen Bausteinen benutzt werden können. Die 8086/8088-Befehle, um den 8255 in diesen Zustand zu bringen, sind:

```
MOV  AL, 10000000B   ;STEUERWORT IN DAS
                     ;AL-REGISTER
OUT  13H, AL         ;GIB STEUERWORT AN DEN
                     ;8255 AUS
```

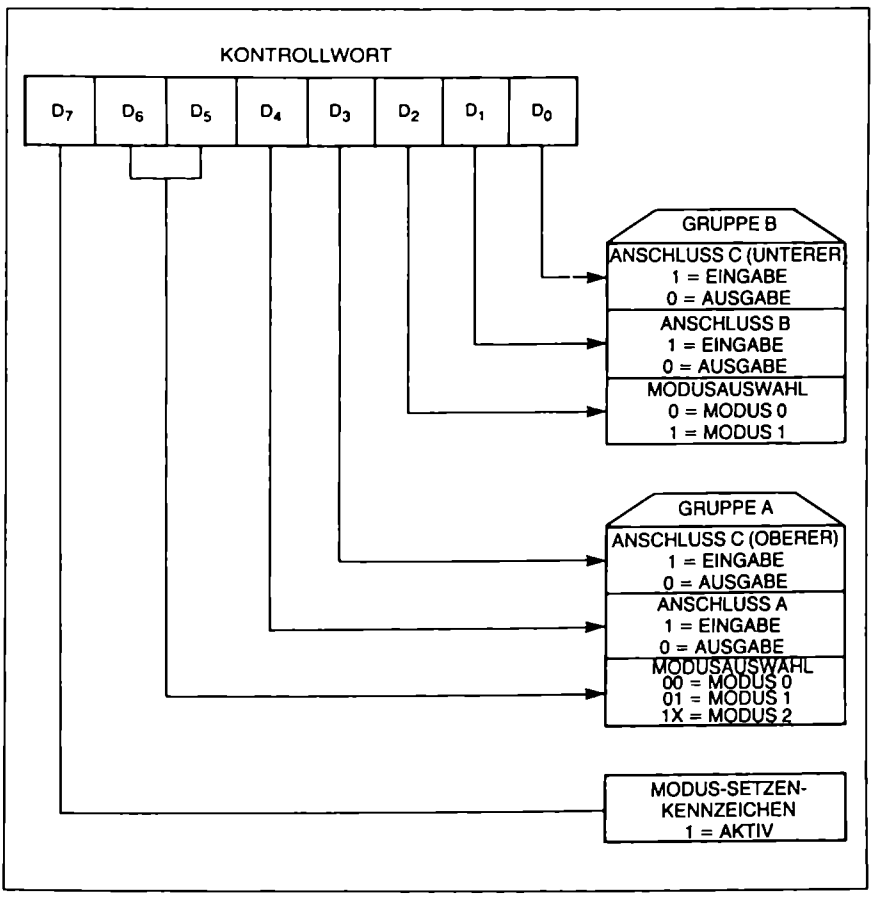


Abb. 7.22: Bit-Definitionen für die Programmierung des Kontrollwortes des 8255

Wenn der 8255 einmal mit diesen beiden Befehlen programmiert worden ist, können wir einfach Daten zu irgendeinem Ausgabeanschluß schreiben, indem wir die 8086/8088-Ausgabebefehle benutzen. Lassen Sie uns zum Beispiel annehmen, daß wir 23H nach Ausgabeanschluß A, 41H nach Ausgabeanschluß B und 73H nach Ausgabeanschluß C schreiben wollen. Um dies zu erreichen, brauchen wir ein Programm wie das folgende:

```
MOV  ALH,23H  ;DATEN FÜR ANSCHLUSS A
OUT  10H,AL   ;GIB DATEN NACH ANSCHLUSS A AUS
MOV  AL,41H   ;DATEN FÜR ANSCHLUSS B
OUT  11H,AL   ;GIB DATEN NACH ANSCHLUSS B AUS
MOV  AL,73H   ;DATEN FÜR ANSCHLUSS C
OUT  12H,AL   ;GIB DATEN NACH ANSCHLUSS C AUS
```

Bei der Ausführung dieser Befehle werden die Ausgabeanschlüsse A, B und C des 8255 mit den angegebenen Datenwerten programmiert. Dies ist eine praktische Möglichkeit, drei einzelne Ausgabeanschlüsse auf einem einzigen Chip vereinigt zu haben.

Wir könnten die Anschlüsse für eine Kombination aus Eingabe und Ausgabe programmieren. Zum Beispiel könnten wir die Anschlüsse A und C zu Ausgabeanschlüssen und Anschluß B zu einem Eingabeanschluß programmieren. In Abbildung 7.22 wäre das Steuerwort, das den 8255 auf diese Konfiguration einstellt:

```
D7                    D0
1  0  0  0  0  0  1  0
```

Nachdem das Steuerwort in Register 3 des 8255 programmiert ist, ist der Baustein logisch (wie in Abbildung 7.23 gezeigt) eingestellt. Ein IN-Befehl könnte verwendet werden, um die Eingabedaten von Anschluß B zu lesen:

```
IN   AL,11H    ;LIES DATEN VON ANSCHLUSS B
```

Wenn der 8255 einmal auf die richtige Konfiguration eingestellt ist, können die Daten leicht an jedem Bausteinanschluß gelesen oder geschrieben werden. Es gibt mehrere verschiedene Kombinationen, den 8255 in einer Modus-0-Standard-E/A-Konfiguration einzustellen. Bis jetzt haben wir nur wenige erwähnt.

EIN BEISPIEL FÜR DAS ABTASTEN EINES TASTENFELDES

Als ein Beispiel für die Benutzung des 8086/8088 mit dem 8255 wollen wir nun eine Tastenfeldschnittstelle behandeln. Lassen Sie uns annehmen, daß das Tastenfeld als eine 4 x 4 Matrix von „single pole, single throw (SPST)"-Schaltern organisiert ist, und lassen Sie uns den 8255 verwenden, um die Tastenfeldschalter mit dem 8088 zu verbinden. Abbildung 7.24 zeigt ein Blockdiagramm dieses Systems.

Die Ausgabeanschlüsse B sind mit den Tastenfeldspalten verbunden. Die Eingabeanschlüsse A sind mit den Tastenfeldreihen verbunden. Die Spaltenleitungen werden jeweils eine nach der anderen auf logisch 0 gesetzt. Nachdem eine Spaltenleitung auf logisch 0 gesetzt wurde, wird der Eingabeanschluß A gelesen. Wenn irgendein von dem Eingabeanschluß gelese-

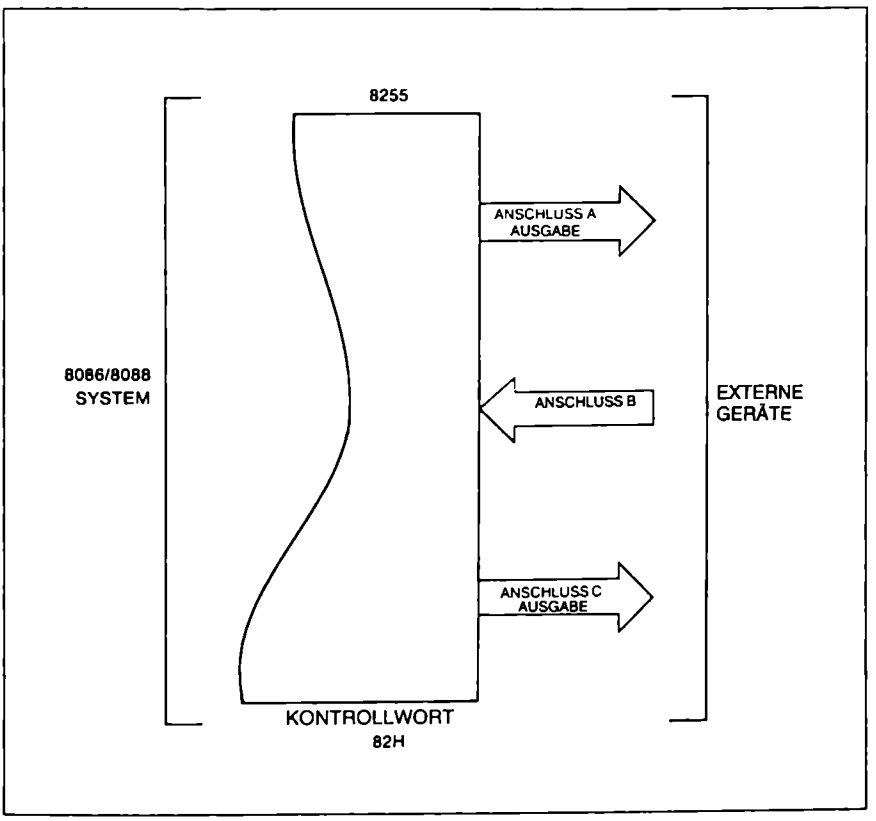


Abb. 7.23: Dieses Blockdiagramm zeigt die Konfiguration des 8255 nach der Ausgabe des Kontrollwortes zum Baustein

nes Bit logisch 0 ist, ist eine Taste gedrückt worden. Ausgehend von der Spaltenleitung, die auf logisch 0 ist, und dem Bit in dem Eingabeanschluß, das logisch 0 ist, können wir sagen, welche Taste in der Matrix gedrückt worden ist. Abbildung 7.25 zeigt ein Programm, das in der gerade beschriebenen Weise arbeitet.

Obwohl dieses Beispiel für eine kleine (4 x 4) Matrix konstruiert wurde, kann es leicht auf eine Matrix beliebiger Größe ausgedehnt werden. Die Verwendung des 8255 ist eine einfache Art, die notwendige Hardware für die Schnittstelle zur Verfügung zu stellen.

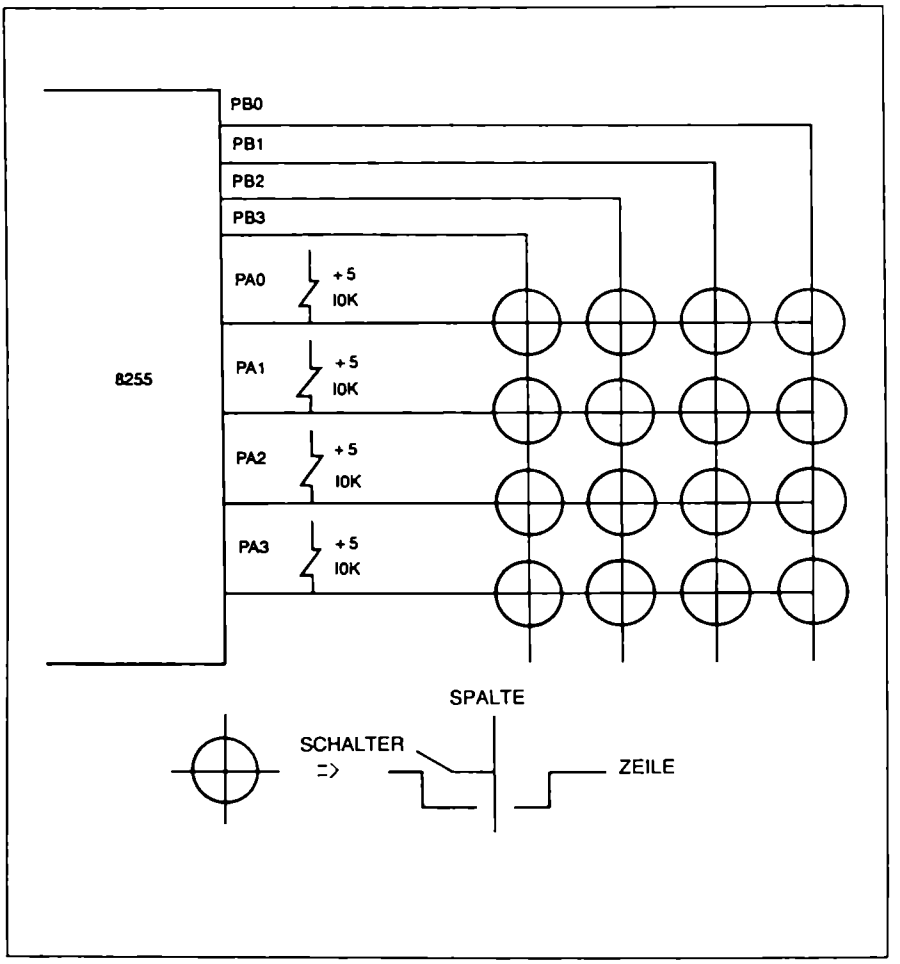


Abb. 7.24: Ein schematisches Diagramm der Verbindungen für ein einfaches 4x4-Tastaturfeld, angeschlossen an einen 8255

```
                    1 ;
                    2 ; EIN PROGRAMM ZUR ERKENNUNG EINES TASTENDRUCKS FÜR
                    3 ; DIE MATRIX IN ABBILDUNG 7.24
                    4 ;
                    5 ; PORT B IST DER AUSGABEANSCHLUSS
                    6 ; PORT A UND PORT C SIND EINGABEANSCHLÜSSE
                    7;
                    8 PORTA     EQU 10H
                    9 PORTB     EQU 11H
                   10 PORTC     EQU 12H
                   11 CONP      EQU 13H          ;KONTROLLANSCHLUSS
                   12 CONWD     EQU 99H          ;KONTROLLWORT
                   13 ;
                   14 ;
0000 B099          15           MOV AL,#CONWD    ;LADE CONWD IN DAS AL-REG
0002 E613          16           OUT CONP,AL      ;CONWD ZUM 8255 KONTR-ANSCH.
                   17 ;
0004 BA1100        18           MOV DX,#PORTB    ;DX = ADRESSE VON PORT B
                   19 ;
0007 B3FE          20 COL       MOV BL,#OFEH     ;SPALTE 0 AKTIV
0009 88D8          21 COL1      MOV AL,BL        ;AKTIVE SPALTE NACH AL
000B EE            22           OUT DX,AL        ;UND GIB AKTIVE SPALTE AUS
000C E410          23           IN AL,PORTA      ;LIES DIE ZEILENLEITUNGEN
000E F6D0          24           NOT AL           ;KOMPLEMENTIERE EINGABEWORT
0010 240F          25           AND AL,#0FH      ;MASKIERE UNBENUTZTE BITS AUS
0012 3C00          26           CMP AL, #00H     ;IRGENDWELCHE AKTIVE 1'EN?
0014 750A          27           JNZ KEYIN        ;FALLS JA, DANN TASTE BETÄT.
0016 D0C3          28           ROL BL,1         ;ROTIERE AKTIVE SPALTE LINKS
0018 80FBEF        29           CMP BL,#0EFH     ;LETZTE SPALTE ERREICHT?
001B 75EC          30           JNZ COL1         ;WENN NEIN, DANN DIE NÄCHSTE
001D E9E7FF        31           JMP COL          ;WIEDERHOLE DEN VORGANG
                   32 ;
                   33 ; DAS FOLGENDE VERARBEITET EINE TASTENBETÄTIGUNG
                   34 ;
0020 88C1          35 KEYIN     MOV CL,AL        ;ZEILENPOSITION=1 INS CL-REG
0022 88D8          36           MOV AL,BL        ;SPALTENPOSITION INS AL-REG
0024 F6D0          37           NOT AL           ;SPALTENPOSITION=1 IM AL-REG
0026 240F          38           AND AL,#0FH      ;UNBENUTZTE BITS = 0
0028 C3            39           RET              ;RÜCKSPRUNG AUS UNTERPROG.
                   40 ;
                   41 ; RÜCKSPRUNG AUS DEM UNTERPROGRAMM MIT
                   42 ; AL-REGISTER = SPALTENPOSITION, 1 IN AKTIVER SPALTE
                   43 ; CL-REGISTER = ZEILENPOSITION, 1 IN AKTIVER ZEILE
                   44 ;
```

Abb. 7.25: Dieses 8086/8088-Programm kontrolliert das 4x4-Tastenfeld, das in Abb. 7.24 gezeigt wurde

Die Programmierung des 8253-Zeitgeberbausteins

In diesem Abschnitt wollen wir zeigen, wie der 8086/8088 zur Steuerung eines programmierbaren Zeitgeberbausteins benutzt werden kann. Ein solcher Baustein wird in vielen Systemanwendungen, in denen Zeitregulierungsaufgaben anliegen, verwendet. Rufen Sie sich ins Gedächtnis zurück, daß wir in einem früheren Abschnitt dieses Kapitels gezeigt haben, wie ein Impuls mit Hilfe von Software erzeugt werden kann. Dies haben wir getan, indem wir den Mikroprozessor in einer Softwareschleife verwendet haben.

Bei der Verwendung eines Zeitgeberbausteins braucht der Mikroprozessor nur einige Steuerbytes zu schreiben, um die Zeitparameter einzustellen, und der Baustein erledigt den Rest der Arbeit. Dies stellt den Mikroprozessor zur Ausführung anderer Arbeiten frei.

Blockdiagramm des 8253-Zeitgeberbausteins

Abbildung 7.26 zeigt ein Blockdiagramm des 8253. In dieser Abbildung können wir erkennen, daß es drei unabhängige programmierbare Zähler gibt. Da alle Zähler identisch sind, werden wir sie nicht getrennt behandeln. Stattdessen wollen wir uns auf die Programmierung eines einzelnen Zählers konzentrieren. Diese Information kann zur Programmierung eines jeden der drei Zähler verwendet werden.

Ein wichtiger Block in Abbildung 7.26 ist der Datenbuspuffer. Dies ist die Logik, die den Systemdatenbus zwischen dem Mikroprozessor und den internen Registern des 8253 puffert. Der Block, der das Lesen und Schreiben der Zählerregister steuert, ist mit Lese-/Schreiblogik bezeichnet.

Der letzte Block in Abbildung 7.26 ist das Steuerwortregister. Dieses Register enthält die programmierte Information, die dem Baustein von dem Systemmikroprozessor gesendet wird. In der Tat definiert dieses Register, wie der Baustein logisch arbeitet. Abbildung 7.27 stellt ein Blockdiagramm vor, das zeigt, wie der 8253 mit dem 8088-System verbunden ist.

Programmierbare Impulserzeugung

Zurückblickend auf das Blockdiagramm in Abbildung 7.27 können wir sehen, daß die Anschlußadressen für den Baustein 30H, 31H, 32H und 33H sind – wobei 33H der Kontrollanschluß ist und 30H , 31H und 32H die Zählregister 0,1 und 2 sind. Lassen Sie uns nun den 8253 programmieren, um einen Impuls bestimmter Dauer unter Programmkontrolle zu erzeugen. Diese Aktion wird manchmal als *Einmal-Impuls* (*one shot*, Monoflop) bezeichnet.

In dem programmierbaren Impulsmodus kann der 8253 so programmiert werden, daß er einen Impuls ausgibt, der einer ganzzahligen Anzahl von

Zeittakten entspricht. Der Einmal-Impuls wird ausgelöst durch die steigende Flanke des Gattereingangs. Falls die Auslösesignale während der Impulsausgabe auftreten, wird der Baustein erneut getriggert und die Impulsausgabe um die volle programmierte Zeitdauer verlängert (retriggerbares Monoflop). Dies ist in Abbildung 7.28 dargestellt.

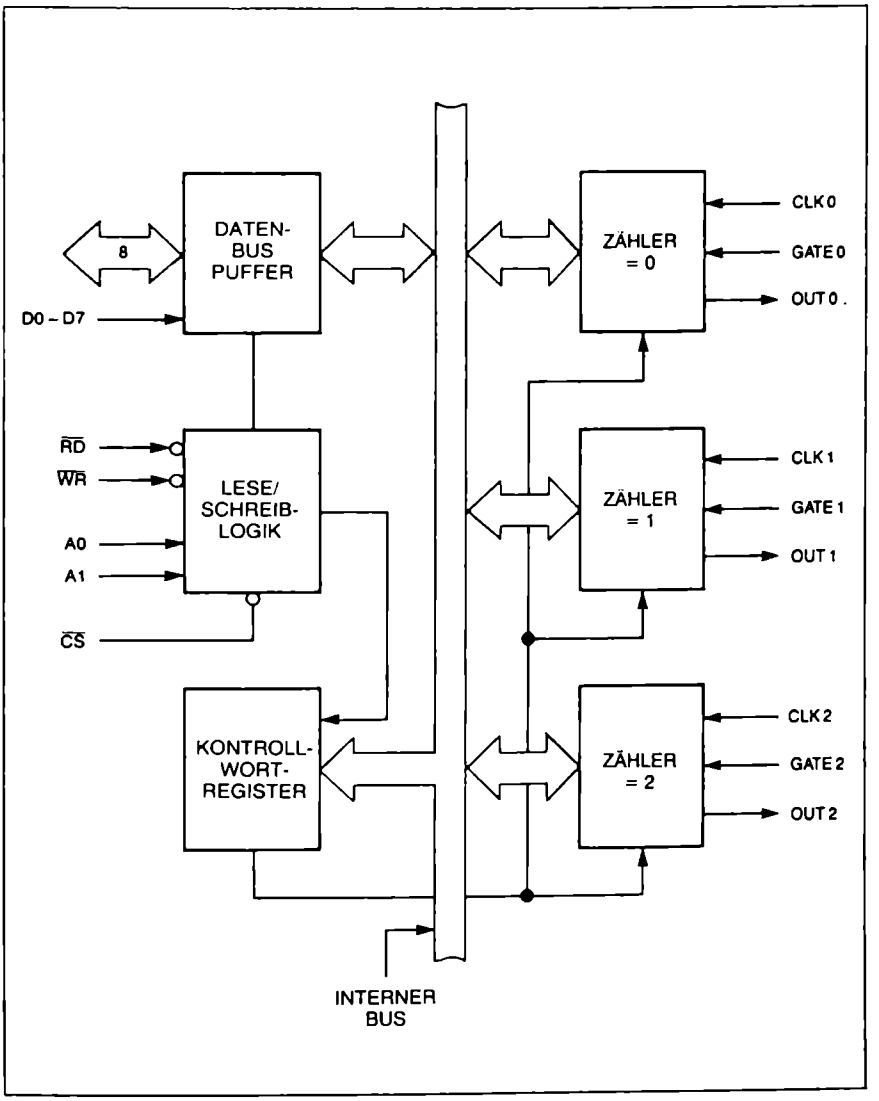


Abb. 7.26: Dieses Blockdiagramm zeigt den programmierbaren Zeitgeberbaustein 8253

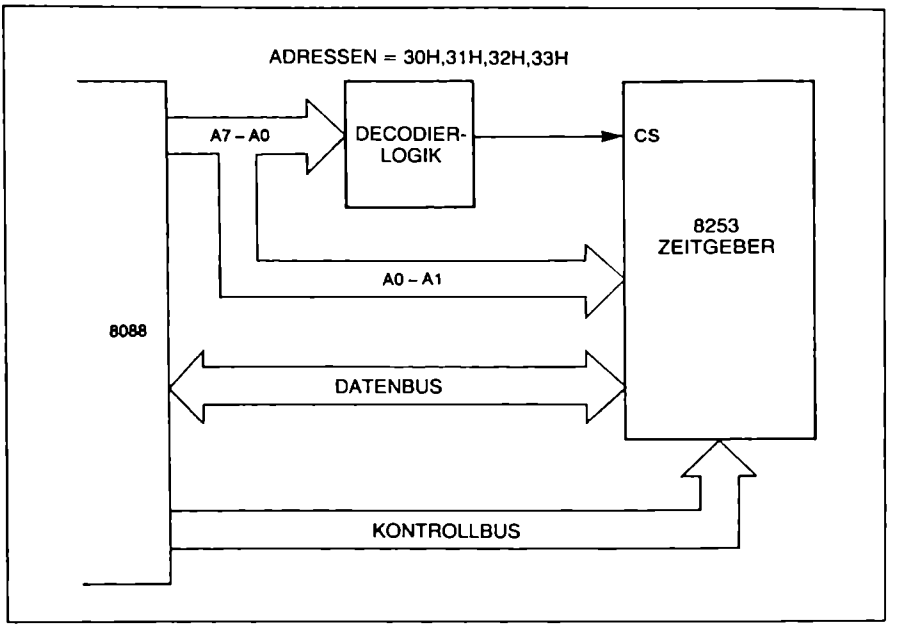


Abb. 7.27: Dieses Diagramm zeigt die Verbindungen des 8253 mit der 8088-CPU

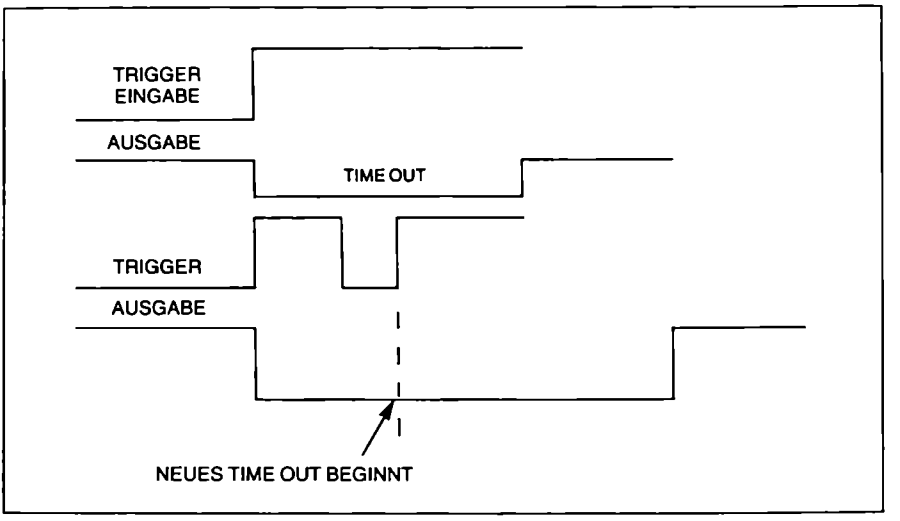


Abb. 7.28: Dieses Zeitdiagramm zeigt, wie die 8253 als programmierbares Monoflop benutzt werden kann. Falls die Triggereingabe auf logisch 1 geht, geht die Ausgabe für die programmierte Zeit des Zählers auf logisch 0. Falls die Triggereingabe während dieser Zeit erneut betätigt wird, verlängert sich das Timeout um eine volle Periode (retriggerbares Monoflop)

Lassen Sie uns annehmen, daß die Frequenz der Eingangstakte des 8253 1 Megahertz beträgt. Wir wollen den 8253 programmieren, einen Impuls von exakt 75 Mikrosekunden auszugeben. Das 8086/8088-Programm, das diese Einstellung vornimmt, ist in Abbildung 7.29 angegeben. In dieser Abbildung nehmen wir an, daß der Zähler Nr. 1 als der programmierbare Einmal-Impulsgeber benutzt wird.

```
                1 ;
                2 ; EIN PROGRAMM ZUR PROGRAMMIERUNG DES 8253
                3 ; ALS EIN PROGRAMMIERBARES MONOFLOP
                4 ;
                5 CNTR1  EQU 31H          ;ADRESSE DES ANSCHLUSSES ZÄHLER 1
                6 CONP   EQU 33H          ;ADRESSE DES KONTROLLANSCHLUSSES
                7 ;
0000 B072       8        MOV AL, #72H     ;KONTROLLWORT NACH AL
0002 E633       9        OUT CONP,AL      ;KONTROLLWORT ZUM 8253
               10 ;
               11 ; CTR 1,RL = 3,M = 1, BINÄRES ZÄHLEN
               12 ;
0004 B04B      13        MOV AL,#75       ;DEZIMALE 75
0006 E631      14        OUT CNTR1,AL;  NIEDERWERTIGES BYTE ZUM ZÄHLER 1
0008 B000      15        MOV AL, #00
000A E631      16        OUT CNTR1,AL;  HÖHERWERTIGES BYTE ZUM ZÄHLER 1
               17 ;
               18 ;
               18 ; DER BAUSTEIN IST NUN EINGESTELLT ZUM WARTEN AUF
               19 ; EINEN AUSLÖSEIMPULS ALS START FÜR DEN ZÄHLER
               20 ;
               21 ;
```

Abb. 7.29: Ein 8086/8088-Programm um den 8253 als programmierbares Monoflop mit einer Impulsdauer von 75 Mikrosekunden einzustellen.

ZUSAMMENFASSUNG

In diesem Kapitel haben wir die wesentlichen Punkte der 8086/8088-Eingabe- und Ausgabeoperationen vorgestellt. Wir haben mit der Definition der Begriffe Eingabe und Ausgabe begonnen. Dann haben wir die beiden 8086/8088-E/A-Befehle IN und OUT behandelt. Wir haben gelernt, daß beide Befehle mit fester und variabler Anschlußadresse verwendet werden können. Beispiele wurden angegeben. Schließlich haben wir zwei Beispiele zur Programmierung spezieller LSI-Peripheriebausteine untersucht – Bausteine, die in vielen 8086/8088-Systemen gebräuchlich sind. Bei der Untersuchung jedes Bausteins haben wir die Anschlußkodierung und ein Blockdiagramm der Verbindung zwischen System und Baustein untersucht.

In Kapitel 8 wollen wir unsere Diskussion der 8086/8088-Anwendungen fortsetzen und Programme für den 8088 schreiben, die mit dem BIOS (Basic Input Output System) des IBM Personal Computers verwendet werden können. Das Verständnis der in diesem Kapitel vorgestellten Konzepte wird Ihnen beim Schreiben der Programme im nächsten Kapitel helfen.

Kapitel 8

Weitere Anwendungen mit dem IBM Personal Computer

In diesem Kapitel setzen wir unsere Diskussion nützlicher Anwendungen des 8086/8088 fort. Wir bereiten zwei vollständige Softwarebeispiele vor. In diesen Beispielen wird Code für den IBM Personal Computer geschrieben – ein System, das den 8088-Mikroprozessor verwendet.

BIOS (BASIC I/O SYSTEM, Grund-E/A-System)

Bevor wir richtige Programme in Assemblersprache vorbereiten, wollen wir unsere Aufmerksamkeit einigen Details des IBM PC zuwenden. Lassen Sie uns insbesondere das BIOS, das Grundsystem für Eingabe und Ausgabe untersuchen. Der Ausdruck BIOS wird häufig in der Architektur und Literatur von Personal Computern verwendet.

Ein System-BIOS bezieht sich auf die Art, mit der das System den Zugang zu den Peripheriegeräten, die der Computer benutzt, zuläßt. Typische Ein-/Ausgabegeräte sind:

1. ein Disketten oder Plattenlaufwerk (hard disk),
2. ein Drucker,
3. ein Bildschirm,
4. eine Eingabetastatur,
5. ein Anschluß für serielle Kommunikation (RS-232) oder ein Fernschreiber (Teletype)

Beachten Sie, daß jeder Computer seine eigene Zusammenstellung von Peripheriegeräten hat, die er steuert.

VORTEILE DES BIOS

Wenn wir die Hardwareschnittstellen zu einigen der oben aufgeführten Peripheriegeräte untersuchen, stellen wir fest, daß diese recht komplex sind. Lassen Sie uns als Beispiel das Diskettenlaufwerk nehmen.

Um die Hardware eines Diskettenlaufwerks (d.h. die Bewegung des Kopfes und die Lese- und Schreibelektronik) zu steuern, sind Kenntnisse über winzige Details im Zusammenspiel zwischen Systemsoftware und Systemhardware erforderlich. Die Kenntnis solcher Details ist für die Personen wichtig, die den Code zur Steuerung des Geräts auf unterster Ebene schreiben. Jedoch für die Person, die nur an der Benutzung des Geräts

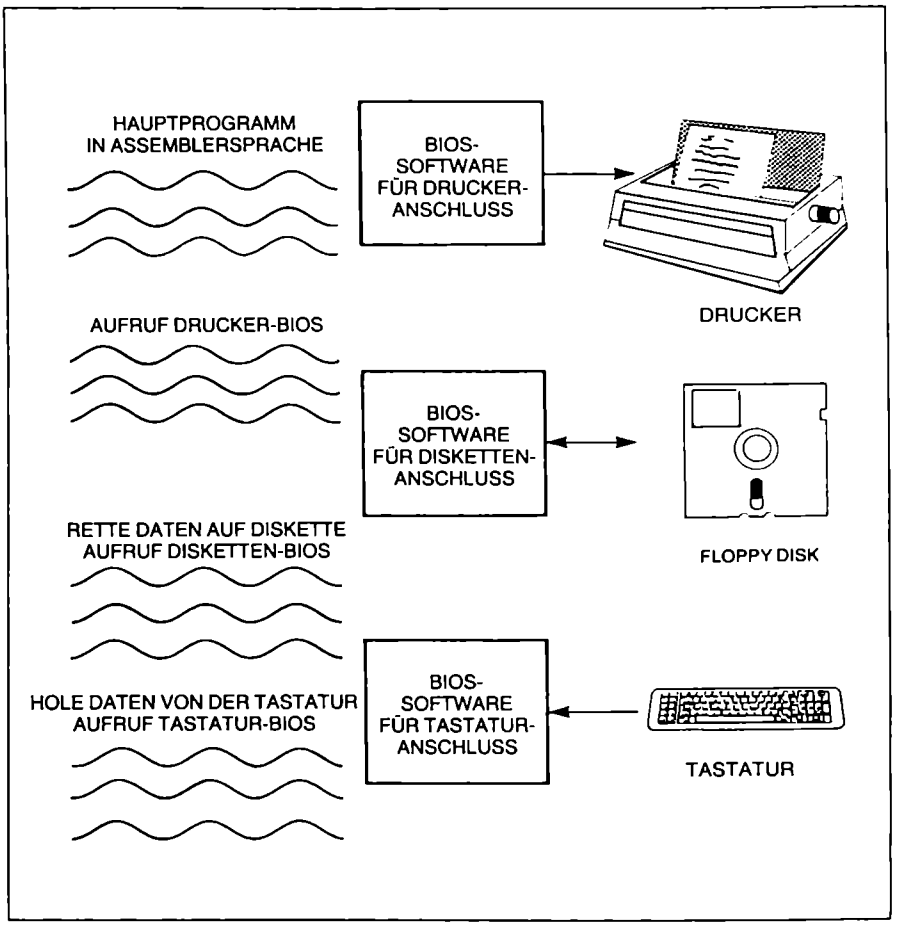


Abb 8.1: Das Hauptprogramm in Assemblersprache kann auf einfache Weise die BIOS-Routinen, die auf der linken Seite gezeigt sind, aufrufen

interessiert ist, ist es eine enorme Belastung, all die schwierigen Details des Systems zu lernen. Das BIOS macht diesen Schritt überflüssig. Das BIOS erlaubt dem Programmierer, Anforderungen auf höherer Ebene nachzukommen (wie zum Beispiel eine Datei von der Diskette zu lesen) und nicht mit den Details zur Erfüllung dieser Anforderung belastet zu sein (wie zum Beispiel zu prüfen, ob der Lesekopf bereit ist, oder die Parameter für den Steuerbaustein einzustellen). Einige Steuerbausteinparameter beeinhalten die Formatierung der Daten auf der Diskette, die festlegt, ob die Daten einfach- oder doppelt-dicht auf der Diskette gespeichert werden usw.

Um zum Beispiel einen Sektor von der Diskette zu lesen, muß der Diskettensteuerbaustein über zahlreiche Details informiert sein, die solche Dinge wie die Anordnung auf der Platte berücksichtigen (d.h. wie viele Bytes/Sektoren einfach oder doppelseitig vorhanden sind, usw.). Mit dem BIOS kann der Programmierer einen Parameter (oder Binärcode) übergeben, indem er einfach angibt, daß ein bestimmter Sektor gelesen werden sollte, und dies dann definiert. Das BIOS führt dann den Befehl und alle Softwarekommandos niedriger Ebenen aus, die zur Durchführung der Anforderung notwendig sind (siehe Abb. 8.1).

DER ZUGRIFF AUF PERIPHERIEGERÄTE MIT DEM BIOS

Jedes System hat eine unterschiedliche Technik, auf das BIOS zuzugreifen, was allgemein in der Systemliteratur beschrieben wird. Beim IBM PC erfolgt der Zugriff auf das BIOS über die Unterbrechungsstruktur der 8088-CPU (die im Kapitel 6 behandelt wurde).

Rufen Sie sich ins Gedächtnis zurück, daß Unterbrechungen mit der 8086/8088-CPU mit Hilfe von Hardware und Software erzeugt werden können. Beim IBM PC können wir, um auf die BIOS-Routine zuzugreifen, Software benutzen und Unterbrechungen erzeugen. Jedes Peripheriegerät, das mit dem IBM PC verbunden ist, hat eine oder mehrere Unterbrechungstypen, die reserviert sind, um es vom Programm aus anzusprechen. Abbildung 8.2 zeigt die 8088-Vektortypen und die entsprechende Hardware, die sie ansprechen.

Die Parameterübergabe an eine BIOS-Routine

Wenn wir eine Routine im BIOS benutzen, müssen sehr oft Parameter vom aufrufenden Programm an die Routine übergeben werden. Solche Parameter können Daten sein, die ausgedruckt werden sollen, eine Baudrate, die für eine RS-232 Verbindung eingestellt werden soll, oder Eingabedaten von einer Tastatur – um nur einige zu nennen. Im IBM BIOS werden alle Daten von und zu der Routine mittels der internen Register des 8088 übergeben. Abbildung 8.3 zeigt diese Handhabung in Blockdiagrammform.

UNTERBRECHUNGSVEKTOREN (0 – 7F)

ADRESSE	UNTER-BRECHUNG	FUNKTION
0-3	0	DIVISION DURCH NULL
4-7	1	EINZELSCHRITT
8-B	2	NICHTMASKIERBARE UNTERBRECHUNG (NMI)
C-F	3	„BREAKPOINT" BEFEHL ('C'CX)
10-13	4	ÜBERLAUF
14-17	5	DRUCKE BILDSCHIRM
18-1F	6,7	RESERVIERT
20-23	8	ZEITGEBER (18.2 PRO SEKUNDE)
24-27	9	TASTATUR-UNTERBRECHUNG
28-37	A,B,C,D	RESERVIERT
38-3B	E	DISKETTEN-UNTERBRECHUNG
3C-3F	F	RESERVIERT
40-43	10	VIDEO-E/A-AUFRUF
44-47	11	GERÄTETEST-AUFRUF
48-4B	12	SPEICHERTEST-AUFRUF
4C-4F	13	DISKETTEN-E/A-AUFRUF
50-53	14	RS-232-E/A-AUFRUF
54-57	15	KASSETTEN-E/A-AUFRUF
58-5B	16	TASTATUR-E/A-AUFRUF
5C-5F	17	DRUCKER-E/A-AUFRUF
60-63	18	ROM BASIS-EINGANGSMODUS
64-67	19	URLADER
68-6B	1A	TAGESZEIT-AUFRUF
6C-6F	1B	KONTROLLVERLAGERUNG BEI TASTE „BREAK":BEM. 1
70-73	1C	KONTROLLVERLAGERUNG BEI ZEITGEBERUNTERBRECHUNG: BEM. 1
74-77	1D	ZEIGER AUF VIDEOINITIALISIERUNGSTABELLE: BEM. 2
78-7B	1E	ZEIGER AUF DISKETTENINITIALISIERUNGSTABELLE: BEM.2
7C-7F	1F	ZEIGER AUF TABELLE(1B) FÜR GRAFISCHEN ZEICHENSATZ FÜR ASCII 128-255. VOREINGESTELLT AUF 0.0

BEMERKUNGEN: (1) INITIALISIERT BEIM EINSCHALTEN ALS ZEIGER AUF EINEN IRET BEFEHL

(2) INITIALISIERT BEIM EINSCHALTEN ALS ZEIGER AUF TABELLEN IM ROM.

Abb. 8.2: Eine Liste der 8088-Unterbrechungstabelle und der entsprechenden Hardware des IBM PC, zu der diese Unterbrechungen Zugang gestatten.

Eine Frage, die an dieser Stelle auftreten mag, ist: „Welche Register werden zur Parameterübergabe benutzt?" Weiter: „Welche Parameter werden übergeben?" Um diese Fragen zu beantworten, wollen wir die Dokumentation einer BIOS-Routine untersuchen. In dieser Dokumentation ist ein Abschnitt enthalten, der mit PRÄAMBEL überschrieben ist. Die PRÄAMBEL ist der Abschnitt der Dokumentation, der genau erklärt,

welche Parameter übergeben werden und innerhalb welcher Parameter das geschieht. Um mehr über die PRÄAMBEL zu lernen, wollen wir das folgende Beispiel untersuchen.

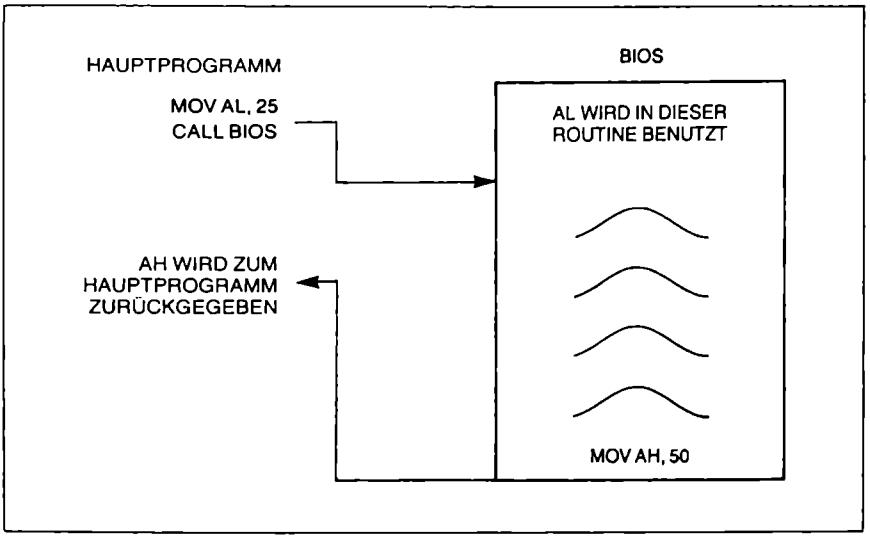


Abb. 8.3: Die Parameterübergabe zum BIOS erfolgt über die internen Register des 8088

PRÄAMBEL FÜR DIE DRUCKER-E/A

BEGINN DER PRÄAMBEL
Die folgende Routine ermöglicht die Kommunikation mit dem Drucker.

(AH) = 0 Drucke das Zeichen in (AL).

Nach der Rückkehr aus dieser Routine ist AH = 1, wenn das Zeichen nicht gedruckt werden konnte; das heißt, wenn eine Zeitsperre erreicht wurde. Die anderen Bits sind wie bei einem normalen Statusaufruf eingestellt.

(AH) = 1 Initialisiere den Druckeranschluß.

Nach der Rückkehr enthält AH den Druckerstatus.

(AH) = 2 Lies den Druckerstatus nach (AH).

Die Bitdefinitionen für den Druckerstatus sind:

D7 = 1 zeigt an, daß der Drucker beschäftigt ist.

D6 = 1 ist das Quittungsbit.

D5 = 1 bedeutet, daß der Drucker kein Papier mehr hat.

D4 = 1 zeigt an, daß der Drucker ausgewählt ist.

D3 = 1 zeigt einen E/A-Fehler an.

D2 = unbenutzt.

D1 = unbenutzt.

D0 = 1 ist eine Anzeige für eine Zeitsperre.

ENDE DER PRÄAMBEL

Aus dieser Präambel des Drucker-BIOS können wir erkennen, daß das AH-Register den Ablauf der Handlung der Routine steuert, wenn diese Routine durch die korrekte Softwareunterbrechung aufgerufen wird. Deshalb müssen wir sicherstellen, daß das AH-Register vor Aufruf der Routine entsprechend gesetzt ist.

Wenn wir aus der Routine zurückkehren, kann das aufrufende Programm den Wert des AH-Registers prüfen, um logisch den Druckerstatus anzuzeigen, das heißt, ob die Routine die beabsichtigte Funktion ausgeführt hat oder nicht.

Ein erstes Beispiel: Die Benutzung der Drucker-BIOS- Routine

In diesem Beispiel schreiben wir eine Routine in 8088-Assemblersprache, die das gerade beschriebene DRUCKER-BIOS benutzt.

Beschreibung des Problems

Bevor wir das eigentliche Programm zur Benutzung des Drucker-BIOS schreiben, wollen wir uns das zu lösende Problem erörtern. In dieser speziellen Anwendung wollen wir Daten, die in einem Softwarepuffer gespeichert sind, drucken. Diese Daten werden in dem Hauptprogramm erstellt. Wenn der Puffer voll ist, werden sie zu dem Drucker-BIOS geschickt, um ausgedruckt zu werden.

In diesem Beispiel nehmen wir an, daß der Puffer gefüllt worden ist, und geben nur die Software an, die nötig ist, um den Pufferinhalt zu drucken. Die Speicherzelle für das erste Element in dem Puffer wird mit BUF_START (BUF wie BUFFER, engl. Puffer) markiert. Das letzte Zeichen in dem Puffer ist dadurch gekennzeichnet, daß es gleich 00H oder das achtzigste Zeichen im Puffer ist. Ein Flußdiagramm für dieses Programm erscheint im nächsten Abschnitt.

Flußdiagramm für das Problem

Abbildung 8.4 zeigt ein Flußdiagramm für eine Lösung dieses Problems.

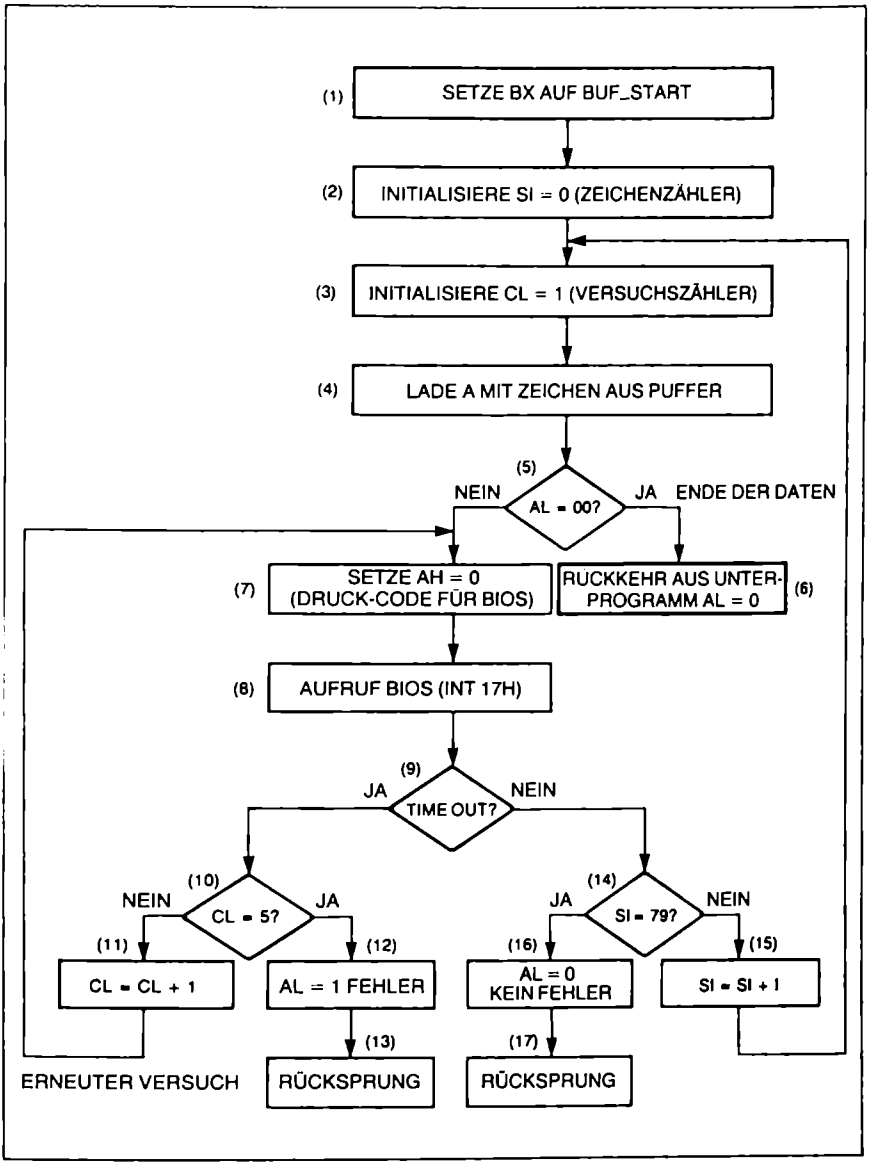


Abb. 8.4: Ein Flußdiagramm zur Ausgabe eines Daten-Puffers mit dem Drucker-BIOS

```
                    ;
                    ;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
                    ;
                    ; BEISPIELROUTINE FÜR DIE BENUTZUNG DES DRUCKER-BIOS
                    ; AUF DEM IBM PC
                    ;
                    ; REG BX = STARTPOSITION DES AUSZUGEBENDEN PUFFERS
                    ;     SI = ZEICHENZÄHLER
                    ;     CL = ZAHL DER VERSUCHE, MAXIMAL 5
                    ;     AL = DEM BIOS ÜBERGEBENES ZU DRUCKENDES ZEICHEN
                    ;     AH = DRUCKCODE 0 - 2, DEM BIOS ZU ÜBERGEBEN
                    ;
                    ; RÜCKSPRUNG AUS DEM UNTERPROGRAMM MIT AL=0: KEIN
                    ; FEHLER,AL=1:FEHLER. DIES WIRD IM HAUPTPROGRAMM
                    ; ÜBERPRÜFT
                    ;
                    ;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
                    ;
                    ; SETZE STARTADRESSE DES UNTERPROGRAMMS AUF 0B00H
                    ;
                            ORG 0B00H             ;SETZE STARTADRESSE
                    ;
                    ; SCHRITT 1
                    ;
0B00 B25B320B               MOV BX,*BUF_START     ;SETZE PUFFERADRESSE IN BX
                    ;
                    ; SCHRITT 2
                    ;
0B03 BE0000                 MOV SI,*0000H         ;SETZE ZEICHENZÄEHLER = 0
                    ;
                    ; SCHRITT 3
                    ;
0B06 B101           STEP3   MOV CL,*01H           ;SETZE VERSUCHSZÄHLER = 1
                    ;
                    ; SCHRITT 4
                    ;
0B08 8A00                   MOV AL,[BX,SI]        ;LADE AL MIT ZEICHEN
                    ;
                    ; SCHRITT 5
                    ;
0B0A 3C00                   CMP AL,*00H           ;LETZTES ZEICHEN????
0B0C 7501                   JNZ STEP7             ;FALLS NEIN, DANN STEP7
                    ;
                    ; SCHRITT 6 DATENENDE,RÜCKSPRUNG MIT AL=0, KEIN FEHLER
                    ;
0B0E C3                     RET                   ;AL = 0, RÜCKSPRUNG
```

Abb. 8.5: Ein 8088-Programm zur Realisierung des Flußdiagramms in Abb. 8.4

```
                47 ;
                48 ; SCHRITT 7   SETZE DEN DRUCKCODE FÜR DAS BIOS = 0
                49 ;
0B0F B400       50 STEP7    MOV AH, #00
                51 ;
                52 ; SCHRITT 8
                53 ;
0B11 CD17       54          INT 17H              ;AUFRUF DER BIOS-ROUTINE
                55 ;
                56 ; SCHRITT 9
                57 ;
0B13 80E401     58          AND AH,#01H          ;PRÜFE AUF ZEITSPERRE
0B16 740D       59          JZ STEP14            ;GEGLÜCKTE ÜBERTRAGUNG
                60 ;
                61 ; SCHRITT 10   ZEITSPERRE WURDE VOM BIOS SIGNALISIERT
                62 ;
0B18 80F905     63          CMP CL,#05           ;SCHON FÜNF VERSUCHE????
0B1B 7405       64          JZ STEP12            ;JA, RÜCKSPRUNG MIT FEHLER
                65 ;
                66 ; SCHRITT 11   NOCH KEINE FÜNF VERSUCHE ERFOLGT
                67 ;
0B1D FEC1       68          INC CL               ;ERHÖHE VERSUCHSZÄHLER
0B1F E9EDFF     69          JMP STEP7            ;VERSUCHE ES NOCH EINMAL
                70 ;
                71 ; SCHRITT 12   FEHLER, RÜCKSPRUNG MIT AL = 1
                72 ;
0B22 B001       73          STEP12               MOV AL,#1
                74 ;
                75 ; SCHRITT 13
                76 ;
0B24 C3         77          RET                  ;RÜCKSPRUNG Z. HAUPTPROGR.
                78 ;
                79 ; SCHRITT 14   KEINE ZEITSPERRE, ZEICHEN GEDRUCKT
                80 ;
0B25 81FE4F00   81 STEP14   CMP SI,#79           ;LETZTES ZEICHEN
0B29 7404       82          JZ STEP16            ;JA, RÜCKSPRUNG MIT AL = 0
                83 ;
                84 ; SCHRITT 15
                85 ;
0B2B 46         86          INC SI               ;ERHÖHE DEN ZEICHENZÄHLER
0B2C E9D7FF     87          JMP STEP3            ;DRUCKE DAS NÄCHSTE ZEICHEN
                88 ;
                89 ; SCHRITT 16   RÜCKKEHR OHNE FEHLER, GEGLÜCKTE AUSGABE
                90 ;
0B2F B000       91 STEP16   MOV AL,#0
```

Abb. 8.5: Ein 8088-Programm zur Realisierung des Flußdiagramms in Abb. 8.4 (Fortsetzung)

```
                    92 ;
                    93 ;SCHRITT 17  RÜCKSPRUNG
                    94 ;
0B31 C3             95              RET
                    96 ;
                    97 ; DER DRUCKPUFFER FOLGT DIREKT AUF DIESE ROUTINE
                    98 ;
                    99 ;
                   100 BUF_START    RSV 80          ;RESERVIERE 80 POSITIONEN
                   101                              ;FÜR DEN PUFFER
0B82 90            102              NOP


TOTAL ERRORS:                       0


SYMBOL             VALUE            LINE #
-------------------------------------------

BUF_START          B32              100
STEP12             B22               73
STEP14             B25               81
STEP16             B2F               91
STEP3              B06               33
STEP7              B0F               50
```

Abb. 8.5: Ein 8088-Programm zur Realisierung des Flußdiagramms in Abb. 8.4 (Fortsetzung)

8088-Programm zur Realisierung des Flußdiagramms

Abbildung 8.5 zeigt ein 8088-Programm, das dem in Abbildung 8.4 gezeigten Flußdiagramm entspricht. Lassen Sie uns dieses untersuchen, so daß wir genau verstehen, wofür jeder Befehl benutzt wird. Bevor wir beginnen, beachten Sie jedoch, daß alle in dem Flußdiagramm von Abbildung 8.4 gekennzeichneten Schritte in dem Programm gezeigt werden. Dies hilft Ihnen zu verstehen, wie jeder Schritt des Flußdiagramms in der mnemotechnischen Darstellung für den 8088 realisiert wird. Die in Abbildung 8.5 verwendete mnemonische Darstellung gilt nur für die 8088-Assemblersprache. Einige der Ausdrücke und Zahlenbezeichner können variieren, wenn Sie eine Assemblersprache von einem anderen Hersteller benutzen.

Schritt 1 In diesem Schritt wird die (zur Übersetzungszeit berechnete) Adresse des Puffers in das BX-Register gebracht. BUF_START ist eine

Marke am Ende des Programms. Es wird unmittelbare Adressierung verwendet, das heißt, der Adreßwert selbst anstelle der Daten in der entsprechenden Speicheradresse wird in das BX-Register gebracht.

Schritt 2 und 3 In den Schritten 2 und 3 werden der Zeichenzähler im SI-Register auf Null und der Zähler für die Anzahl der Versuche im CL-Register auf Eins gesetzt. Der Zeichenzähler verfolgt die Position des Zeichens im Puffer, das gerade gedruckt wurde. Für dieses Register wird ein maximaler Wert von 79 verwendet. Das resultiert aus der Tatsache, daß die Zählung bei Null beginnt.

Das CL-Register, das die Anzahl der Versuche zählt, wird verwendet, wenn ein Zeitsperrenfehler von der BIOS-Routine zurückgegeben wird. Es werden maximal fünf Versuche unternommen, um das Zeichen zu drucken. Wenn das Zeichen nach fünf Versuchen nicht gedruckt wurde, kehrt das Unterprogramm in das Hauptprogramm mit einer 1 im AL-Register zurück und zeigt so einen Fehler in der Druckroutine an.

Schritt 4 In diesem Schritt wird das Zeichen von dem Puffer in das AL-Register gebracht. Beachten Sie, daß die Adressierungsart indirekt mit einer Basis plus Index ist. Das Basisregister ist BX und das Indexregister ist SI. Rufen Sie sich ins Gedächtnis zurück, daß SI der Zeichenzähler und BX die Startadresse des Puffers ist.

Schritt 5 In diesem Schritt untersuchen wir die Daten im Puffer. Wenn ein Wert gleich 00 ist, zeigt das das Ende der zu druckenden Daten an, und wir kehren aus dem Unterprogramm (Schritt 6) mit AL = 0 zurück, was das erfolgreiche Ausdrucken des Puffers signalisiert. Wenn AL nicht gleich 00 ist, wird das Zeichen gedruckt (Schritt 7).

Schritt 6 In diesem Schritt kehren wir aus diesem Unterprogramm in das Hauptprogramm zurück. Es sollte bemerkt werden, daß dieser Rücksprung ein NEAR-Intrasegment-Rücksprung ist. Die Art des Rücksprungs bestimmt, wie Daten vom Stapel heruntergeholt werden (siehe Kapitel 5).

Schritte 7 und 8 Diese zwei Schritte stellen den Druckcode für die BIOS-Routine ein und rufen dann die Routine auf. Der Aufruf erfolgt mit INT 17H. Dies ist eine Softwareunterbrechung.

Schritt 9 Wenn das Programm diesen Punkt erreicht, ist es aus dem BIOS zurückgekehrt. Wir decken die oberen Bits des AH-Registers ab, um das Zeitsperrenbit D0 auf eine logische 1 zu überprüfen. Wenn dieses Bit logisch 0 ist, fahren wir mit Schritt 14 fort. Wenn es logisch 1 ist, ist ein Zeitsperrenfehler aufgetreten. Wenn dies der Fall ist, fahren wir mit Schritt 10 fort.

Schritt 10 und 11 Diese Schritte werden benutzt, um zu prüfen, ob wir fünfmal versucht haben, dasselbe Zeichen auszugeben. Wenn das der Fall

ist, fahren wir mit Schritt 12 fort. Wenn nicht, springen wir nach Schritt 7 zurück, nachdem wir das CL-Register erhöht haben. Das zu druckende Zeichen ist noch im AL-Register. Nur das AH-Register muß vor dem Aufruf der BIOS-Routine neu initialisiert werden. Dies resultiert aus der Tatsache, daß die Drucker-BIOS-Routine nur den Inhalt von AH verändert.

Schritt 12 und 13 In Schritt 12 laden wir 01 in das AL-Register. Dies zeigt dem aufrufenden Hauptprogramm an, daß ein Fehler in der Pufferdruckroutine aufgetreten ist. Schritt 13 kehrt in das Hauptprogramm zurück.

Schritt 14 Wenn wir Schritt 14 erreichen, ist ein einzelnes Zeichen erfolgreich vom Drucker-BIOS gedruckt worden. Wir prüfen nun, ob dies das letzte zu druckende Zeichen war oder nicht; das heißt: ist SI=79? Wenn das der Fall ist, ist das letzte Zeichen aus dem Puffer gedruckt worden, und wir springen nach Schritt 16 zurück und setzen das AL-Register gleich 0. Die Null zeigt an, daß der Puffer erfolgreich ausgedruckt wurde.

Schritt 15 Schritt 15 erhöht das SI-Register, damit dieses auf das nächste zu druckende Zeichen im Puffer zeigt. Nach der Erhöhung des Zeigers springt das Programm nach Schritt 3.

Schritt 16 und 17 Diese beiden Schritte setzen das AL-Register gleich Null und kehren dann zum Hauptprogramm zurück. Eine Null im AL-Register zeigt an, daß der Ausdruck des Puffers erfolgt ist.

Der Druckpuffer

Am Ende des in Abbildung 8.5 gezeigten Unterprogramms sehen wir, daß der Druckpuffer 80 Bytes reserviert hat. Der Puffer beginnt an der Speicheradresse 0B32H, welches die Adresse direkt nach dem RET-Befehl ist. Das Ende des Puffers ist an der Speicheradresse 0B81H, was gleich 0B32H + 79 dezimal ist und zu insgesamt 80 Speicherzellen führt. Die Startadresse des Puffers ist mit BUF_START markiert.

Beachten Sie, daß die Symboltabelle am Ende des Programms erscheint.

Ein anderes Beispiel mit Tastatur- und RS-232-BIOS

Lassen Sie uns ein zweites Beispiel ansehen. In diesem Beispiel wollen wir ein Programm vorbereiten, das das RS-232- BIOS und das Tastatur-BIOS benutzt. Dieses Programm liest einen Tastendruck von der Tastatur ein und gibt ihn über die RS-232- Kommunikationsleitung wieder aus. Diese Art von Anwendungsprogramm könnte als Teil eines größeren Programms benutzt werden, das den IBM PC in die Lage versetzt, als einzelnes Terminal zu fungieren.

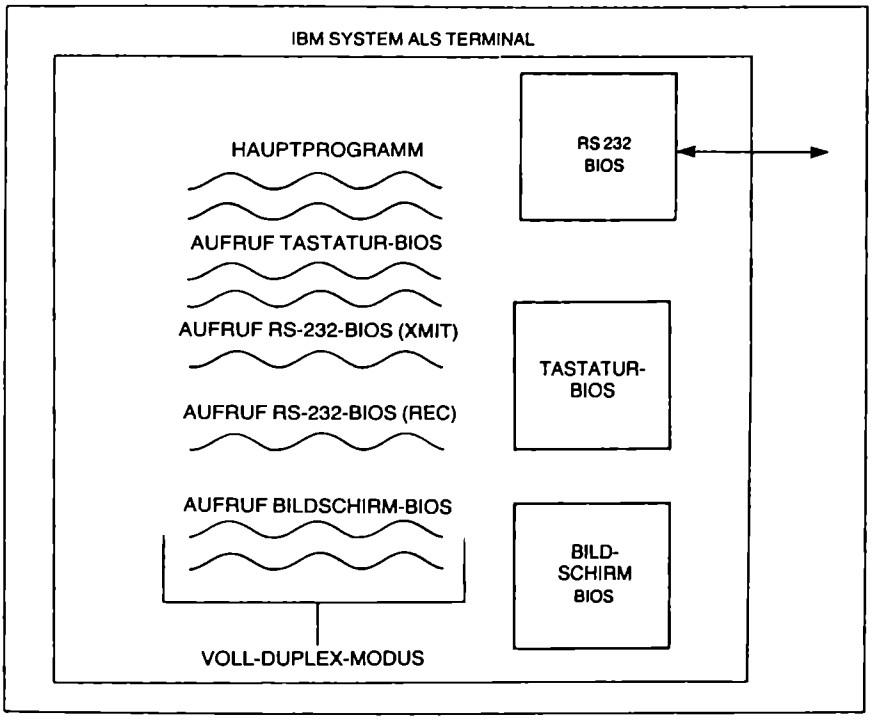


Abb. 8.6: Um den IBM PC als Terminal zum Anschluß an andere Systeme zu benutzen, benötigen Sie die oben gezeigten BIOS-Routinen

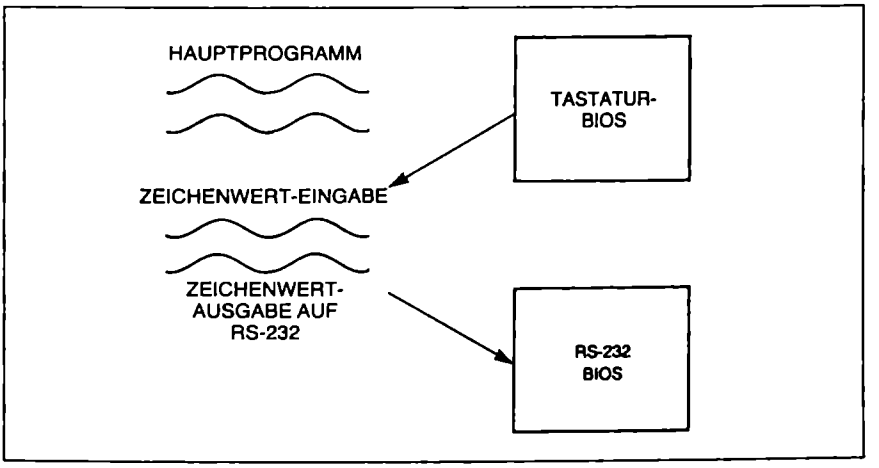


Abb. 8.7: In diesem Programm werden nur das RS-232-BIOS und das Tastatur-BIOS benutzt. Wir lesen ein Zeichen von der Tastatur ein und geben es auf die RS-232-Kommunikationsbaugruppe aus

In solch einer Anwendung erhielten Sie Zeichen von dem RS-232-Anschluß und gäben sie auf dem Bildschirm des Terminals aus. Abbildung 8.6 zeigt ein Blockdiagramm der Routinen, die notwendig sind, um den IBM PC als Terminal zu betreiben.

Abbildung 8.7 zeigt ein Blockdiagramm der beiden Operationen, die wir in dieser speziellen 8088-Programmanwendung ausführen.

TASTATUR-PRÄAMBEL

Bevor wir von dem Tastatur-BIOS Gebrauch machen können, müssen wir verstehen, wie Parameter von und zu der Routine übergeben werden. Diese Information wird in der folgenden PRÄAMBEL des Tastatur-BIOS geliefert:

BEGINN DER PRÄAMBEL

Tastatur E/A INT 16H

(AH) = 0 liest das nächste an der Tastatur eingetippte ASCII-Zeichen; gibt dann das Ergebnis in das AL-Register und den Abtastcode in das AH-Register zurück.

(AH) = 1 setzt das Z-Kennzeichenbit, um anzuzeigen, ob ein ASCII-Zeichen zum Lesen verfügbar ist.

ZF) = 1 zeigt an, daß kein Code verfügbar ist.

(ZF) = 0 zeigt an, daß kein Code verfügbar ist. Wenn ZF gleich Null ist, steht das nächste zu lesende Zeichen im Puffer, im AX-Register, und der Eintrag bleibt in dem Puffer.

(AH) = 2 gibt den aktuellen Zustand der Umschalttasten in das AL-Register zurück. Die Biteinstellungen für diesen Code werden in den EQU-Anweisungen für die KB-Kennzeichenbits angezeigt. Diese Bitzuordnungen sind:

```
INS_STATE     EQU  80H;  EINFÜGESTATUS IST AKTIV
CAPS_STATE    EQU  40H;  DER ZUSTAND DER
                         ARRETIERUNGSTASTE FÜR
                         GROSSBUCHSTABEN HAT SICH
                         UMGEKEHRT

NUM_STATE     EQU  20H;  DER ZUSTAND DER TASTE FÜR
                         DEN NUMERISCHEN MODUS
                         HAT SICH UMGEKEHRT
```

```
SCROLL_STATE  EQU  10H;  DER ZUSTAND DER TASTE FÜR
                        DEN SCROLL-MODUS HAT SICH
                        UMGEKEHRT
CTL_SHIFT     EQU  04H;  DIE TASTE FÜR DEN CTL-
                        MODUS WURDE BETÄTIGT
LEFT_SHIFT    EQU  02H;  DIE LINKE SHIFT-TASTE WURDE
                        BETÄTIGT (GROSSSCHREIBUNG)
RIGHT_SHIFT   EQU  01H;  DIE RECHTE SHIFTTASTE
                        WURDE BETÄTIGT
                        (GROSSSCHREIBUNG)
```

Nur das AX-Register und die Kennzeichenbits werden von dieser Routine beeinflußt. Alle anderen Register behalten ihren Wert.

ENDE DER TASTATUR-PRÄAMBEL

Mit dieser Präambel sind wir in der Lage, die Tastaturroutine zur Eingabe eines Tastenanschlags zu benutzen.

RS-232-PRÄAMBEL

Wir benutzen das RS-232-BIOS in diesem Beispiel. Hier ist die PRÄAMBEL für das RS-232-BIOS.

BEGINN DER RS-232-PRÄAMBEL

RS232_IO INT 14H

Diese Routine stellt Bytestrom-E/A zum Kommunikationsanschluß zur Verfügung gemäß den Parametern:

(AH) = 0 Initialisiere den Kommunikationsanschluß. AL enthält die Bits für die Initialisierung.
Diese Bits werden so definiert:

BITS	7,6,5	Baudrate
	0 0 0	110
	0 0 1	150
	0 1 0	300
	0 1 1	600
	1 0 0	1200
	1 0 1	2400
	1 1 0	4800
	1 1 1	9600

Bits	4,3	Paritätsbit
	x 0	keins
	0 1	ungerade
	1 1	gerade
Bit	2	Anzahl der Stoppbits
	0	1
	1	2
Bits	1,0	Wortlänge
	1 0	7 bit
	1 1	8 bit

Bei der Rückkehr aus einer Routine, die mit AH = 0 aufgerufen wurde, sind die Zustandsbits so gesetzt, wie nach einem Aufruf des Kommunikationsstatus (AH = 3), wie unten angegeben.

(AH) = 1 sendet das Zeichen in AL über die Kommunikationsleitung. Das AL-Register bleibt erhalten.

Beim Verlassen der Routine wird Bit 7 von AH gesetzt, falls die Routine nicht in der Lage war, das Datenbyte zu übermitteln. Der Rest von AH ist wie in einem Statusaufruf gesetzt und gibt so den Zustand der Leitung wieder.

(AH) = 2 empfängt ein Zeichen in (AL) von der Kommunikationsleitung, bevor der Rücksprung zum aufrufenden Programm erfolgt.

Beim Verlassen der Routine enthält AH den aktuellen Zustand der Leitung, wie er von der Statusroutine gesetzt wird, mit der Ausnahme, daß nur die Fehlerbits (7, 4, 3, 2, 1) einen gültigen Wert haben. In diesem Fall zeigt das Zeitsperrenbit, daß das „data set ready"-Signal der RS−232 Leitung nicht empfangen wurde. AH ist nur ungleich Null, wenn ein Fehler aufgetreten ist.

(AH) = 3 gibt den Zustand des Kommunikationsanschlusses nach (AX) zurück.

AH enthält den Leitungskontrollstatus:

bit 7 = Zeitsperre

bit 6 = Sender bereit

bit 5 = Sendepuffer leer

bit 4 = „Break" entdeckt

bit 3 = Zeichenfehler

bit 2 = Paritätsfehler

bit 1 = Überlauffehler

bit 0 = Daten verfügbar

AL enthält den Modemstatus:

bit 7 = Empfangssignalpegel

bit 6 = Ankommender Ruf

bit 5 = Betriebsbereitschaft

bit 4 = Sendebereitschaft

bit 3 = Delta-Empfangsbereitschaft

bit 2 = Folgeflanke Ankommender Ruf

bit 1 = Delta-Betriebsbereitschaft

bit 0 = Delta-Sendebereitschaft

AX wird gemäß den Aufrufparametern geändert. Alle anderen Register bleiben unverändert.

ENDE DER RS-232-PRÄAMBEL

PROGRAMMDEFINITION

Unter Benutzung der Präambeln für das RS-232- und das Tastatur-BIOS wollen wir nun ein 8088-Programm zum Einlesen eines Zeichens von der Tastatur und zu dessen Übermittlung über die RS-232-Kommunikationsleitung entwickeln. Das Programm endet, wenn ein Semikolon auf der Tastatur eingegeben wird.

Die Daten werden zu einem anderen System mit folgenden Parametern der seriellen Übermittlung gesendet:

Baudrate	= 2400
Parität	= ungerade
Anzahl der Stoppbits	= 2
Wortlänge	= 8 bit

Vor der Übermittlung irgendwelcher Daten werden diese Parameter am IBM PC RS-232-Anschluß eingestellt.

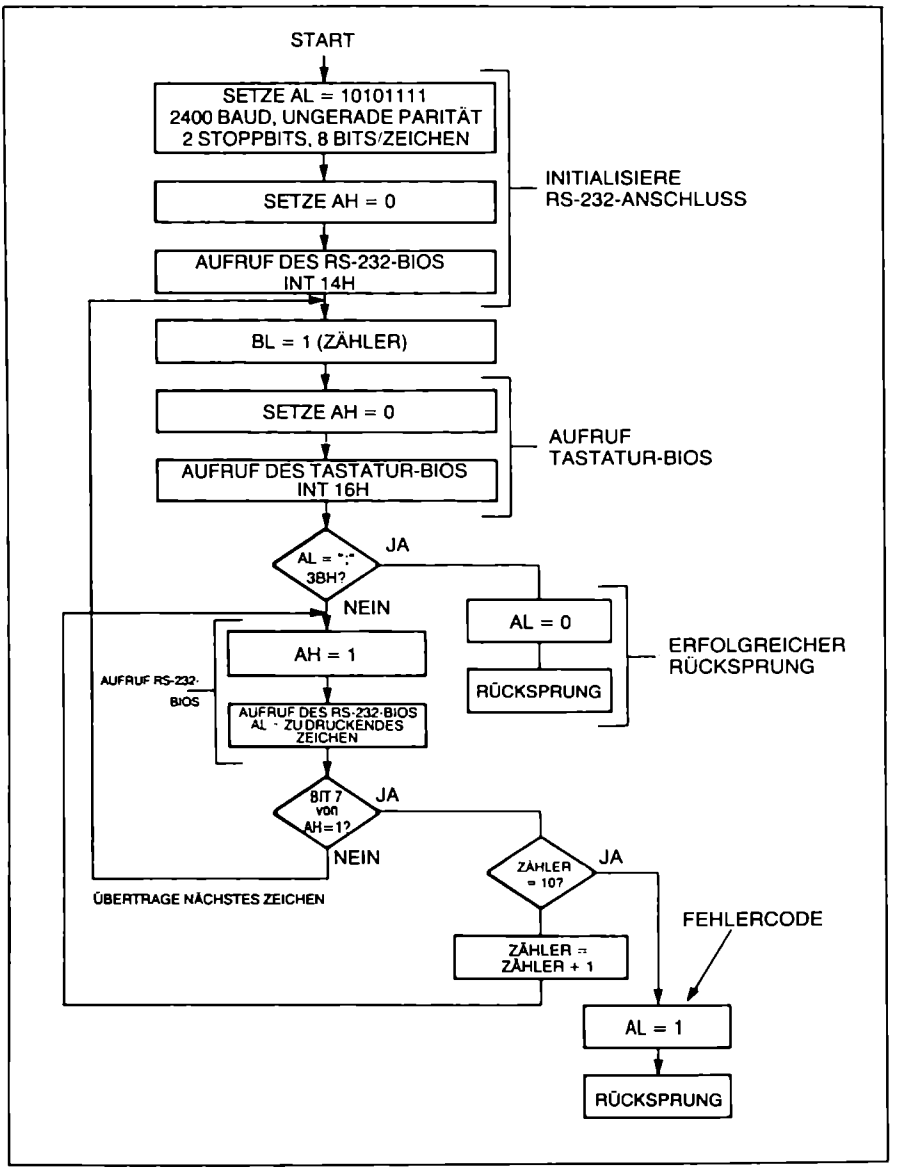


Abb. 8.8: Ein Flußdiagramm zum Lesen eines Zeichens von der Tastatur und zum anschließenden Senden des Zeichens über die RS-232-Kommunikationsleitung

```
                ;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
                ;
                ; UNTERPROGRAMM ZUR BENUTZUNG DES TASTATUR- UND RS-232
                ; BIOS FÜR DEN IBM PERSONAL COMPUTER
                ; KEYBOARD-BIOS = INT 16H
                ; TASTATUR-BIOS = INT 14H
                ;
                ; BL = ANZAHL DER ÜBERTRAGUNGSVERSUCHE BIS ZU
                ;         EINEM METER
                ; AL ZURÜCKGEGEBEN = 0 : KEIN FEHLER BEI ÜBERTRAGUNG
                ; AL ZURÜCKGEGEBEN = 1 : FEHLER BEI ÜBERTRAGUNG
                ;
                ; DAS UNTERPROGRAMM LIEST EIN ZEICHEN VON DER
                ; TASTATUR UND ÜBERTRÄGT DANN DIESES ZEICHEN ÜBER DEN
                ; RS-232 BUS. EIN SEMIKOLON BEENDET DAS UNTERPROGRAMM.
                ;
                ;
                ;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
                ;
                               ORG 0B00H
                ;
                ;              EQU-DIREKTIVEN FÜR DAS UNTERPROGRAMM
                ;
                RS_232         EQU 14H       ;UNTERBRECHUNGSTYP FÜR RS-232
                KEY_BOARD      EQU 16H       ;UNTERBRECHUNGSTYP FÜR TASTATUR
                MAX_COUNT      EQU 10        ;MAXIMALANZAHL VON VERSUCHEN
                SEMI_COLON     EQU 3BH       ;ASCII-DARSTELLUNG DES SEMIKOLONS
                ;
                ;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
                ;
                ; INITIALISIERE DEN RS-232-ANSCHLUSS
                ;
0B00 B0AF                 MOV AL,#10101111P  ;2400 BAUD, UNGERADE PARITÄT
                ;
0B02 B400                 MOV AH,#00         ;SETZE CODE FÜR RS-232
0B04 CD14                 INT RS_232         ;AUFRUF DES RS-232-BIOS
                ;
                ; VERSUCHE NUN, EIN ZEICHEN VON DER TASTATUR ZU LESEN
                ;
                ; ALS ERSTES INITIALISIERE DEN VERSUCHSZÄHLER
                ;
0B06 B300       GO1       MOV BL,#00         ;ZÄHLER = 0
0B08 B400                 MOV AH,#00         ;CODE FÜR TASTATUR-BIOS
0B0A CD16                 INT KEY_BOARD      ;AUFRUF DES TASTATUR-BIOS
                ;
                ; ÜBERPRÜFE ZEICHEN AUF SEMIKOLON
                ;
```

Abb. 8.9: Ein 8088-Programm zur Realisierung des Flußdiagramms aus Abb. 8.8

```
B0C 3C3B      48  CMP AL,# SEMI_COLON        ,ÜBERPRÜFE AUF SEMIKOLON
0B0E 7503     49  JNZ TRANSMIT               ;WENN UNGLEICH, SENDE ZEICHEN
              50;
              51;   FALLS EIN SEMIKOLON VORLIEGT; STOPP DAS PROGRAMM
              52;
0B10 B000     53      MOV AL,#00             ;SETZE AL = 0 : KEIN FEHLER
0B12 C3       54      RET                    ;RÜCKSPRUNG ZUM HAUPTPROGRAMM
              55;
              56;   ÜBERTRAGUNGSTEIL DES UNTERPROGRAMMS
              57;
0B13 B401     58  TRANSMIT MOV AH,#01H       ;SETZE CODE FÜR RS-232-BIOS
0B15 CD14     59      INT RS_232             ;AUFRUF DES RS_232-BIOS
0B17 F6C480   60      TEST AH,#80H           ;IST BIT 7 GESETZT ???
0B1A 74EA     61      JZ GO1                 ;NICHT GESETZT, SO NOCH EINMAL
              62;
              63;   ZEICHEN KONNTE NICHT ÜBERTRAGEN WERDEN
              64;
0B1C 80FB0A   65      CMP BL,#MAX_COUNT      ;SCHON ZEHN VERSUCHE??
0B1F 7405     66      JZ ERROR_END           ;JA, RÜCKSPRUNG MIT FEHLER
0B21 FEC3     67      INC BL                 ;ERHÖHE DEN VERSUCHSZÄHLER
0B23 E9EDFF   68      JMP TRANSMIT           ;STARTE NÄCHSTEN VERSUCH
              69;
              70; FEHLERRÜCKSPRUNG
              71;
0B26 B001     72 ERROR_END MOV AL,#01H       ;SETZE FEHLERCODE
0B28 C3       73      RET
              74;

TOTAL ERRORS:    0

SYMBOL              VALUE      LINE#
-------------------------------------

ERROR_END            B26        72
GO1                  B06        42
KEY_BOARD             16        25
MAX_COUNT              A        26
RS_232                14        24
SEMI_COLON            3B        27
TRANSMIT             B13        58
```

Abb. 8.9: Ein 8088-Programm zur Realisierung des Flußdiagramms als Abb. 8.8 (Fortsetzung)

Flußdiagramm für das Programm

Abbildung 8.8 zeigt ein Flußdiagramm für dieses Programm. Lassen Sie uns dieses Flußdiagramm erörtern, bevor wir es mit der 8088-Software

realisieren. Der erste Schritt ist die Initialisierung des RS-232-Anschlusses.

Als nächstes ruft das Programm das Tastatur-BIOS auf. Wenn ein Zeichen von dem BIOS zurückgegeben wird, wird es dem RS-232-BIOS übergeben. An diesem Punkt wird das Zeichen aus dem IBM PC hinaus über den RS-232-Bus ausgegeben. In der Übermittlungsschleife des Flußdiagramms versuchen wir die Ausgabe des Zeichens bis zu 10mal. Falls das Zeichen nach 10 Versuchen nicht erfolgreich übermittelt worden ist, wird ein Fehlercode gesendet.

Um das Programm zu beenden, tippen Sie ein Semikolon. Das Flußdiagramm in Abbildung 8.6 prüft vor der Übermittlung, ob ein Semikolon vorliegt.

8088-Programm

Ein vollständiges, assembliertes 8088-Programm zur Realisierung des Flußdiagramms in Abbildung 8.8 zeigt Abbildung 8.9.

ZUSAMMENFASSUNG

Wir haben in diesem Kapitel zwei vollständige Beispiele von Anwendungsprogrammen für die 8088-CPU angegeben. Diese Programme haben das spezielle BIOS (Basic Input Output System), das im IBM PC enthalten ist, benutzt. Wenn Sie Zugang zu einem IBM PC haben, sollten diese Beispiele Ihnen beim Einstieg zum Schreiben Ihrer eigenen 8086/8088-Anwendungen helfen.

Kapitel 9

Programmentwicklung

Bis hierhin haben wir in diesem Buch den internen Aufbau der CPU untersucht; wir haben die Benutzung des 8086/8088-Befehlssatzes erörtert und haben Beispiele assemblierter Programme angegeben. In diesem abschließenden Kapitel wollen wir das wichtige Thema der Programmentwicklung behandeln. Bei der Entwicklung von Programmen sind mehrere Ebenen von Hardware- und Softwarebetriebsmitteln zu betrachten. Die Bestimmung der geeigneten Ebene hängt von der speziellen Anwendung ab. In diesem Kapitel stellen wir verschiedene verfügbare Betriebsmittel vor und bewerten sie.

PROGRAMMIERUNGSMÖGLICHKEITEN

Ein Programm kann in Binär- oder Hexadezimalcode, in einer Assemblersprache oder einer höheren Programmiersprache geschrieben werden. Abbildung 9.1 zeigt die verschiedenen Ebenen der Programmierung. Wir wollen sie nun erörtern.

Hexadezimale Verschlüsselung

Einige Programme werden unter Verwendung mnemonischer Zeichen der Assemblersprache geschrieben. Die eigentliche Übersetzung solcher mnemonischer Darstellung in den entsprechenden Binärcode erfordert einen Assembler. Wenn kein Assembler zur Verfügung steht, muß die Übersetzung von Hand erfolgen. Da die Übersetzung in Binärcode langwierig und vor allem fehleranfällig ist, verwenden Benutzer statt dessen oft hexadezimalen Code. Es gibt einige Einplatinencomputer, die nur Programme in hexadezimalem Code akzeptieren. Im allgemeinen ist die Übersetzung von Befehlen in den entsprechenden Code nur für eine kleine Anzahl von Befehlen vernünftig, und diese Anzahl variiert von Programmierer zu Programmierer.

Letztendlich ist die hexadezimale Codierung kein wünschenswerter Weg, einen Computer zu programmieren – es ist einfach kein wirtschaftlicher Weg. Die Kosten für einen Assembler und die erforderliche alphanumerische Tastatur sind verschwindend im Vergleich zu dem erhöhten Aufwand an Zeit und Anstrengung, der nötig ist, um ein Programm direkt in

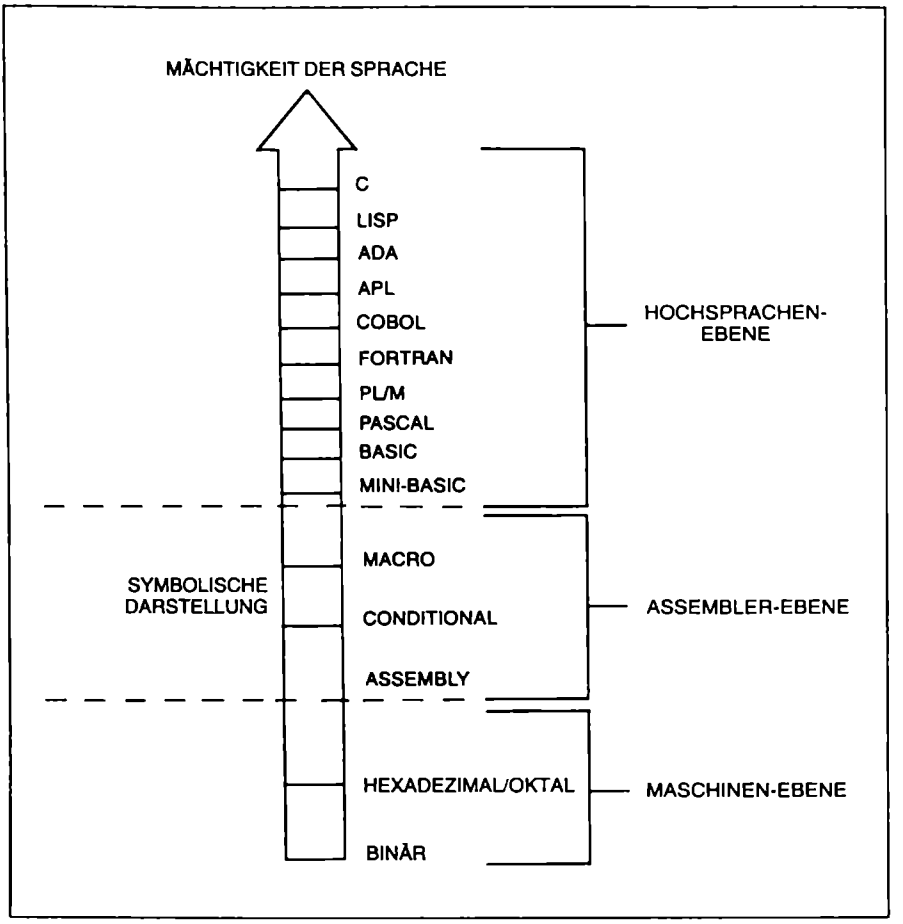


Abb. 9.1: Ein Blockdiagramm, das die verschiedenen Ebenen der Programmierung für ein Mikroprozessorsystem zeigt

den Systemspeicher einzugeben. Wenn es notwendig ist, hexadezimale Codierung zu verwenden, ist es deshalb vernünftig, das Programm in mnemonischen Zeichen der Assemblersprache zu schreiben und es erst dann in hexadezimalen Code zu übersetzen. Dies sollte man immer so machen, weil ein in Assemblersprache geschriebenes Programm leichter zu verstehen und zu testen ist.

Programmieren in Assemblersprache

Programmierung auf Assemblerebene oder in Assemblersprache umfaßt beides, solche Programme, die in das System in hexadezimaler Form ein-

gegeben werden, und solche, die in symbolischer Form der Assemblersprache eingegeben werden. Wir wollen nun die direkte Eingabe eines Programms in seiner Assemblerdarstellung untersuchen.

Wenn ein Programm in Assemblersprache eingegeben wird, muß ein Assemblerprogramm zur Verfügung stehen, das die mnemonischen Befehle des Programms liest. Die Aufgabe eines Assemblers ist es, die Befehle in die entsprechenden Bitmuster zu übersetzen, die 1 bis 6 Bytes umfassen können, jenachdem was bei der Verschlüsselung der Befehle spezifiziert ist. Ein guter Assembler bietet auch eine Reihe zusätzlicher Einrichtungen zur Erstellung eines Programms an.

Später in diesem Kapitel werden wir einen Überblick über die verschiedenen Softwarebetriebsmittel geben, die einem System normalerweise zur Verfügung stehen. Aber zuerst lassen Sie uns eine dritte Programmierungsalternative untersuchen: das Programmieren in einer höheren Programmiersprache.

Programmieren in einer höheren Programmiersprache

Ein 8086/8088-Programm kann auch in einer höheren Programmiersprache wie BASIC, PL/M-86 oder Pascal geschrieben werden, um nur einige zu nennen. Eine höhere Programmiersprache bietet leistungsfähige Befehle an, die das Programmieren schneller und leichter als in Assemblersprache machen. Aber Befehle einer höheren Programmiersprache sind unabhängig von dem verwendeten Prozessor und müssen deshalb innerhalb des Computers von einem komplexen Programm in die endgültige binäre Darstellung übersetzt werden, die der Mikroprozessor ausführen kann.

Typischerweise wird jeder Befehl einer höheren Programmiersprache in viele einzelne binäre Befehle übersetzt, und zwar von einem Programm, das Übersetzer (Compiler) genannt wird, oder von einem Interpretierer (Interpreter). Ein *Übersetzer* übersetzt alle Befehle eines Programms in Objektcode. Im Gegensatz dazu interpretiert ein *Interpretierer* einen einzelnen Befehl, führt ihn aus und übersetzt dann den nächsten und führt ihn aus.

Die Sprache BASIC verwendet einen Interpretierer während die Sprache Pascal einen Übersetzer benutzt. Ein Interpretierer bietet den Vorteil interaktiver Antwort. Dies ermöglicht es dem Programmierer, die Befehle während der Eingabe in den Computer einzeln zu prüfen. Weiter kann ein einzelner Befehl ausgeführt werden, sobald er in den Computer eingegeben ist. Ein Nachteil eines Interpretierers ist die geringe Geschwindigkeit, mit der er Befehle ausführt. Ein übersetztes Programm läuft viel schneller als ein interpretiertes – der Geschwindigkeitsunterschied variiert jedoch von System zu System.

Softwareunterstützung

Wir wollen unsere Untersuchung der Softwareunterstützung mit einem Überblick über die Hauptsoftwarehilfsmittel, die in einem vollständigen System zur Verfügung stehen, beginnen. Später werden wir die vorher eingeführten Definitionen zusammenfassen. Weiter definieren wir wichtige Programme, die in einem Softwareentwicklungssystem verfügbar sind.

Wir wollen mit dem Assembler beginnen. Der *Assembler* übersetzt die mnemonische Darstellung von Befehlen in ihr binäres Äquivalent. Er übersetzt normalerweise einen symbolischen Befehl in einen binären Befehl, der ein bis sechs Datenbytes einnimmt. Der resultierende Binärcode, der *Objektcode* genannt, ist direkt vom Mikroprozessor ausführbar. Ein Assembler erstellt auch eine vollständige Liste der mnemonischen Befehle, ebenso wie eine Liste der definierten Symbole des Programms, wie in Abbildung 9.2 dargestellt. Ein Assembler führt ebenfalls auch syntaktische Fehler, wie fehlerhaft geschriebene oder nichtzulässige Befehle, Verzweigungsfehler oder doppelt verwendete bzw. fehlende Marken auf.

Der *Monitor* ist das Hauptprogramm zur Benutzung der Hardwarebetriebsmittel des Systems. Er überwacht die Eingabeeinrichtungen ständig in bezug auf Eingabe; zusätzlich bringt er die Geräte in die Grundeinstellung zurück. Ein Kleinstmonitor für einen Einplatinencomputer, der mit einer Tastatur und LEDs ausgestattet ist, überprüft zum Beispiel ständig die Tastatur im Hinblick auf Eingaben vom Benutzer. Wenn Eingaben festgestellt werden, gibt der Monitor die entsprechenden Daten auf den LED-Anzeigen aus.

Zusätzlich zu den eben erwähnten Aufgaben muß das Monitorprogramm eine begrenzte Anzahl von Kommandos von der Tastatur erkennen. Beispiele dieser Kommandos sind solche, die den Verarbeitungsablauf eines Programms starten, stoppen, fortsetzen sowie Speicherzellen laden und untersuchen. In einem großen System, das über eine komplexe Dateiverwaltung oder Prozeßsteuerung verfügt, wird der Monitor oft als Ausführungsprogramm eingestuft. Die Obermenge der Hilfsmittel wird *Betriebssystem* genannt. Wenn die Dateien auf Magnetplatte stehen, wird das Betriebssystem ein Plattenbetriebssystem (DOS, disk operating system) genannt.

Ein *Editor* vereinfacht die Eingabe oder Veränderung von Text oder Programmen. Er ermöglicht dem Benutzer, auf bequeme Weise Buchstaben einzugeben, anzuhängen oder in den Text einzufügen. Weiter ermöglicht der Editor dem Benutzer, Textzeilen einzufügen und zu löschen, ebenso wie Buchstaben oder Zeichenfolgen zu suchen. Der Editor ist ein wichtiges Hilfsmittel zur bequemen und effizienten Eingabe von Quellcode.

Der Ausdruck *Debugger (Testhilfeprogramm)* verweist auf ein Hilfsmittel, das bei der Fehlerbeseitigung benutzt wird. Wenn ein Programm beim

```
                 1 ;
                 2 ; EIN PROGRAMM ZUR PROGRAMMIERUNG DES 8253
                 3 ; ALS EIN PROGRAMMIERBARES MONOFLOP
                 4 ;
                 5 CNTR1    EQU 31H          ;ADRESSE DES ANSCHLUSSES ZÄHLER 1
                 6 CONP     EQU 33H          ;ADRESSE DES KONTROLLANSCHLUSSES
                 7 ;
0000 B072        8          MOV AL,#72H      ;KONTROLLWORT NACH AL
0002 E633        9          OUT CONP,AL      ;KONTROLLWORT ZUM 8253
                10          ;
                11 ; CTR 1,RL = 3,M = 1, BINÄRES ZÄHLEN
                12 ;
0004 B04B       13          MOV AL,#75       ;DEZIMALE 75
0006 E631       14          OUT CNTR1,AL;    NIEDERWERTIGES BYTE ZUM ZÄHLER 1
0008 B000       15          MOV AL,#00
000A E631       16          UT CNTR1,AL;     HÖHERWERTIGES BYTE ZUM ZÄHLER 1
                17 ;
                18 ;
                18 ; DER BAUSTEIN IST NUN EINGESTELLT ZUM WARTEN AUF
                19 ; EINEN AUSLÖSEIMPULS ALS START FÜR DEN ZÄHLER
                20 ;
                21 ;

TOTAL ERRORS:                    0

SYMBOL              VALUE          LINE#
-------------------------------------------
CNTR1                 31              5
CONP                  33              6   <------- SYMBOLTABELLE
```

Abb. 9.2: Eine vollständige Liste eines assemblierten Programms. Die Symboltabelle wird am Ende der Assemblierung ausgegeben

erstenmal nicht richtig arbeitet, kann sich ein Testhilfeprogramm als nützlich erweisen. Es erlaubt dem Benutzer, Unterbrechungspunkte einzufügen, um die Ausführung eines Programms zur Untersuchung von Registern und/oder Speicherzellen an genau angegebenen Adressen zu unterbrechen.

Ein *Lader* oder *Bindelader* plaziert verschiedene Blöcke von Objektcode an bestimmte Speicherpositionen und stellt ihre jeweiligen symbolischen Zeiger ein, so daß sie aufeinander verweisen können.

Ein *Simulator* oder ein *Emulator*programm simuliert die Arbeitsweise eines Geräts, das üblicherweise der Mikroprozessor ist, wenn ein Pro-

gramm für ein von einem Mikroprozessor gesteuerten System entwickelt wird. Wenn wir diesen Weg wählen, ist es möglich, das Programm zu unterbrechen, es zu verändern und im RAM-Speicher zu behalten. Ein Nachteil eines Simulators ist, daß seine Ausführungsgeschwindigkeit langsamer als die eines normalen Mikroprozessors ist. Deshalb ist es nicht möglich, Echtzeit-Geräte zu testen. Dies sind Geräte, deren Arbeitsweise davon abhängt, daß das Mikroprozessorsystem mit normaler Geschwindigkeit arbeitet.

Ein Emulator ist im wesentlichen ein Simulator in Echtzeit. Ein Emulator verwendet einen Prozessor, um einen anderen zu simulieren.

Dienstleistungsroutinen sind Routinen mit allgemeinem Zweck, zu denen jeder Benutzer Zugang hat. Das sind z. B. solche Routinen, die Daten im Speicher untersuchen oder eine Eingabe von der Tastatur interpretieren. Diese Routinen sind nicht schwierig zu schreiben, aber da der Benutzer normalerweise von ihnen Gebrauch macht, stellt sie der Hersteller zur Verfügung. Abbildung 9.3 zeigt ein Speicherbild für ein typisches Entwicklungssystem.

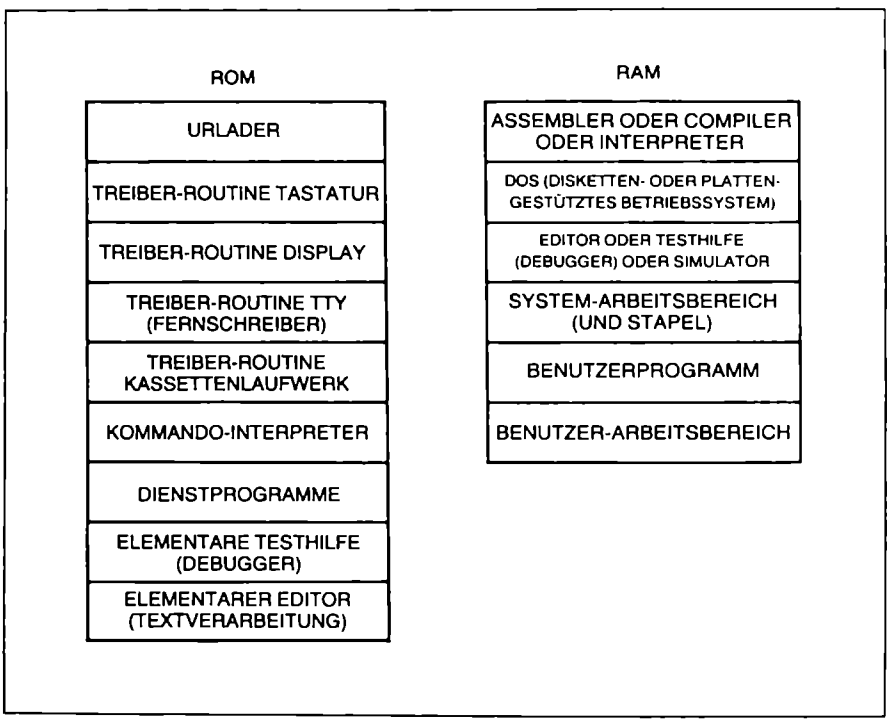


Abb. 9.3: Eine Speicherübersicht für ein typisches Entwicklungssystem, das die verfügbaren Software-Pakete zeigt

DIE STUFEN DER SOFTWAREENTWICKLUNG

Wir untersuchen nun das typische Vorgehen zur Entwicklung eines Programms auf Assemblerebene. Wir nehmen an, daß alle üblichen Softwarehilfsmittel zur Verfügung stehen, damit wir ihren Nutzen demonstrieren können. Wenn sie in einem bestimmten System nicht vorhanden sind, können wir immer noch Programme erstellen, aber die Bequemlichkeit nimmt ab. Auch der Zeitaufwand zur Programmerstellung nimmt wahrscheinlich zu.

Rufen Sie sich ins Gedächtnis zurück, daß der normale Weg, ein Programm auf Assemblerebene zu entwickeln, darin besteht, einen Algorithmus für das zu lösende Problem zu entwerfen. Als nächstes ist es wichtig, eine umfassende Menge von Flußdiagrammen zu entwickeln, die den Programmablauf darstellen. Schließlich werden die Flußdiagramme in die Assemblersprache für den Mikroprozessor übersetzt. Wenn das Programm geschrieben ist, besteht der nächste Schritt darin, es in den Computer einzugeben. Ein Programm kann in den RAM-Speicher des Systems unter der Kontrolle des Editors geladen werden.

Bevor das Programm getestet wird, muß es der Assembler in Binärcode übersetzen. Wenn der Assembler noch nicht in dem System ist, muß ihn der Programmierer von einem externen Speichermedium, wie zum Beispiel einer Platte, laden.

Der Assemblierungsprozeß liefert ein Objektprogramm, das ausgeführt werden kann. Nach dem Assemblierungsprozeß liegen die Befehle in Maschinen- oder Objektcode vor. Die Maschine oder der Mikroprozessor kann diese Art von Code direkt ausführen.

Als nächstes kann das Testhilfeprogramm zur Verifizierung der Arbeitsweise eines Programms benutzt werden, um an kritischen Stellen zur Überprüfung von Zwischenergebnissen eine Reihe von Unterbrechungspunkten zu setzen.

Um das Programm auszutesten, kann ein Emulator verwendet werden. Dieser Emulator ist mit einem Kabel ausgerüstet, das mit einem Stecker mit 40 Anschlußstiften endet, deren Stiftbelegung identisch mit der des 8086/8088 ist. Wenn wir diesen Stecker in das Anwendungssystem, das wir entwickeln, stecken, sind die von dem Emulator erzeugten Signale exakt so, wie die des 8086/8088, nur vielleicht etwas langsamer.

Der Vorteil, einen Testemulator zu verwenden, besteht darin, daß das Programm im RAM (Schreib-/Lesespeicher) bleiben kann, was Änderungen einfacher macht, als befände sich das Programm in EPROMs (einer Art von ROM). Weiter kann der Emulator mit allen E/A-Geräten des Entwicklungssystems kommunizieren. Dieser Vorteil und die Tatsache, daß alle Hilfsmittel des Systems zur Fehlersuche zur Verfügung stehen, machen den Testemulator zu einem sehr attraktiven Hilfsmittel bei der

Programmentwicklung. Damit ist unsere Beschreibung der üblichen Folge von Ereignissen bei der Programmentwicklung komplett. Wir wollen nun einen Überblick über die Hardwarealternativen für die Programmentwicklung geben.

HARDWAREALTERNATIVEN

Es gibt viele verschiedene Hardwaresysteme zur Programmentwicklung, und diese Systeme variieren in bezug auf Preis und Fähigkeiten. In der Regel ist ein System je teurer, je mehr Hilfsmittel es zur Programmentwicklung zur Verfügung stellt.

Einplatinen-Mikrocomputer

Ein Einplatinen-Mikrocomputer bietet den preisgünstigsten Zugang zur Programmentwicklung. Er ist normalerweise mit einer hexadezimalen Tastatur plus einiger Funktionstasten und LEDs ausgestattet, die Informationen anzeigen. Die meisten Einplatinen-Mikrocomputer verfügen nicht über einen Assembler. Einplatinen-Mikrocomputer haben einen kleinen Monitor mit begrenzten Testhilfe- und Editiermöglichkeiten. Wegen dieser Beschränkungen werden Programme in hexadezimaler Form eingegeben und in dieser Form auch auf den System-LEDs angezeigt.

Ein Einplatinen-Mikrocomputer hat theoretisch die gleiche Hardware und Leistung wie jeder andere Computer. Wegen seiner beschränkten Speicherkapazität und der kleinen Tastatur kann er jedoch nicht die üblichen Erleichterungen eines größeren Systems bieten. Dies macht die Programmentwicklung zu einem viel langsameren Prozeß. Wegen der Beschränkungen bei der Programmentwicklung ist ein Einplatinen-Mikrocomputer bestens als Trainings- und erzieherisches Werkzeug geeignet. Einplatinen-Computer bieten wahrscheinlich die kostengünstigste Möglichkeit, programmieren durch die Praxis zu lernen.

Das Entwicklungssystem

Das *Entwicklungs*system ist ein Mikrocomputersystem, das speziell dazu entworfen ist, die Entwicklung von Programmen zu einer möglichst leichten Aufgabe zu machen. Es ist üblicherweise mit den folgenden Komponenten ausgestattet:

1. ein großer RAM-Speicher,
2. eine CRT-Anzeige,
3. ein Drucker,
4. eine Platten-E/A,

5. ein PROM-Progammiergerät,
6. ein Testemulator (in-circuit emulator) (manchmal).

Ein Entwicklungssystem bietet normalerweise alle oder die meisten der im vorigen Abschnitt aufgezählten Softwaremöglichkeiten an. Im besten Fall ist es das ideale Softwareentwicklungswerkzeug.

Hobby-Mikrocomputer

Die Hobby-Mikrocomputer-Hardware ist ähnlich zu der eines Entwicklungssystems. Seine Softwaremöglichkeiten sind jedoch nicht so hochentwickelt wie die eines Entwicklungssystems für industrielle Anwendungen. Solch ein System ist das im vorigen Abschnitt beschriebene. Beispielsweise bieten die meisten Hobby-Mikrocomputer nur elementare Assembler und minimale Editor- und Dateisysteme an. Diese Systeme stellen einen Zwischenschritt zwischen dem Einplatinen-Mikrocomputer und dem vollständigen Mikroprozessorentwicklungssystem dar. Für einen Benutzer, der Programme geringer Komplexität entwickeln möchte, sind diese Systeme wahrscheinlich der beste Kompromiß. Selbst mit ihren Beschränkungen bieten sie noch die Vorteile eines niedrigen Preises und eines vernünftigen Angebots an Softwareentwicklungswerkzeugen.

ZUSAMMENFASSUNG DER HARDWAREBETRIEBSMITTEL

Wir können drei große Kategorien von Hardwaresystemen unterscheiden:

1. Einplatinen-Mikrocomputer,
2. vollständige Entwicklungssysteme,
3. Hobby-Mikrocomputer.

Ein Einplatinen-Mikrocomputer steht denjenigen zur Verfügung, die ein minimales Budget haben und programmieren lernen wollen. Diese Art von System können Sie zur Entwicklung sowohl aller in diesem Text angegebenen als auch vieler anderer Programme benutzen. Bei der Entwicklung von Programmen, die länger als ein paar hundert Zeilen sind, werden Sie den Einplatinen-Mikrocomputer ungeeignet finden.

Ein vollständiges Entwicklungssystem steht Benutzern aus der Industrie zur Verfügung. Jedes Projekt, dem kein vollständiges Entwicklungssystem zur Verfügung steht, wird dies mit erhöhtem Programmentwicklungsaufwand bezahlen. Der „springende Punkt" ist klar: Hardwarebetriebsmittel gegenüber Programmierzeit. Wenn die entwickelten Programme einfach sind, gibt es natürlich weniger teure Möglichkeiten; aber

wenn die Programme komplex sind, ist es schwer, Hardwareeinsparungen zu rechtfertigen, da die Programmierungskosten im allgemeinen die dominierenden Kosten eines Projekts sind.

Ein Hobby-Mikrocomputersystem bietet genügend, wenn auch minimale Hilfsmittel. Dies ändert sich jedoch. Sehr gute Softwareentwicklungssysteme stehen bald für Personal Computer zur Verfügung.

DER ASSEMBLER

Lassen Sie uns nun ein sehr nützliches Entwicklungswerkzeug detaillierter untersuchen: den Assembler. Ein Assembler erlaubt die angenehme symbolische Darstellung eines Benutzerprogramms. Aus dieser symbolischen oder mnemonischen Darstellung erzeugt der Assembler den Binär- oder Objektcode der Befehle.

Assemblerfelder

Beim Tippen eines Programms für den Assembler werden drei Felder benutzt:

1. ***Das Markenfeld*** (Dieses Feld ist fakultativ und kann die symbolische Adresse des folgenden Befehls enthalten.)

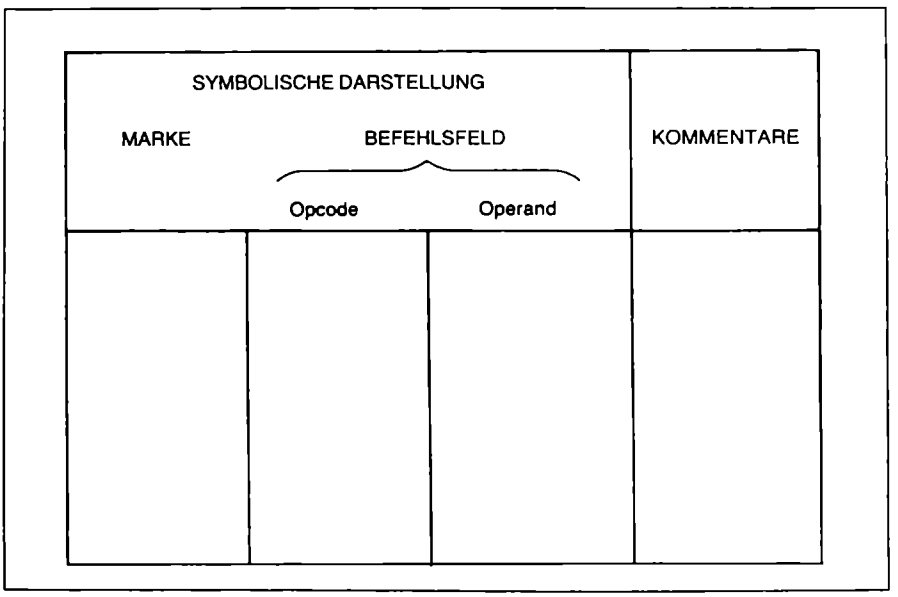


Abb. 9.4: Die Assemblerfelder von links nach rechts sind Marke, Befehl und Kommentar

2. ***Das Befehlsfeld*** (Dieses Feld umfaßt den Operationscode und mögliche Operanden.)

3. ***Das Kommentarfeld*** (Dieses Feld ist fakultativ und ist dazu angelegt, das Programm leichter lesbar und verständlicher zu machen.)

Abbildung 9.4 zeigt diese Felder in einem Diagramm.

Sobald ein Programm dem Assembler eingegeben ist, wird ein Ausdruck produziert. Bei der Erzeugung des Ausdrucks werden drei neue Felder an der äußersten linken Spalte des Papiers oder der Anzeige hinzugefügt.

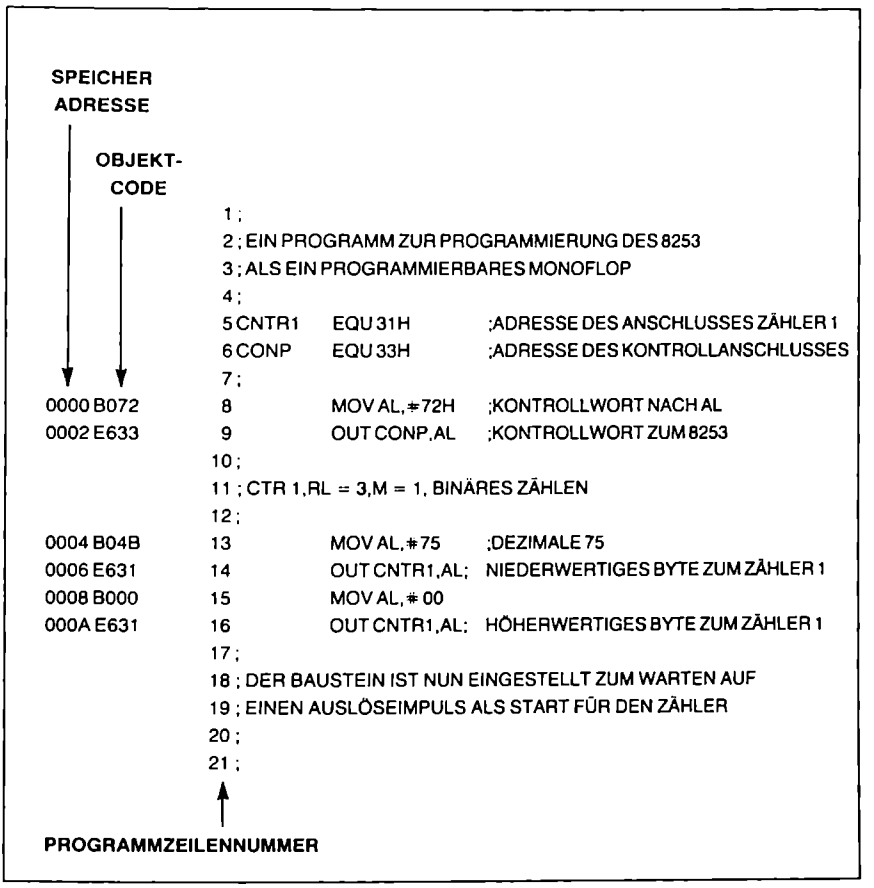


Abb. 9.5: Drei neue Felder werden während des Assemblierungsprozesses erzeugt. Diese Felder sind die Speicheradresse, der Objektcode und die Programmzeilennummer

Diese neuen Felder sind die Speicheradresse, die Bytes des Objektcodes und die Zeilennummer. Abbildung 9.5 zeigt in einem Beispiel eines Assemblerausdrucks diese neuen Felder.

Tafeln

Wenn der Assembler die mnemonischen Zeichen in den Objektcode übersetzt, übersetzt er auch die Symbole, die für Adressen und Konstanten verwendet werden, in ihr binäres Äquivalent. Um die Fehlersuche im Programm zu erleichtern, zählt der Assembler jedes Symbol auf und zeigt, wo es in dem Programm verwendet wird. Dies ist die *Symboltabelle*. Abbildung 9.6 zeigt ein Beispiel einer Symboltabelle.

SYMBOL	VALUE	LINE #
ACCUM	2C1D	1611
ADDREG	F4	36
ALPHA	30D	535
ALPHA1	317	540
ALPHA2	31E	544
AX_REG	2C31	1615
BI	20	39
BLKRAT	3E8	24
BO	10	40
BOOT	7D2	1423
BOOT1	7D5	1424
BP_REG	2C3B	1620
BREAK	6D4	1299
BRKDAT	2C2B	1613
BRKFLG	2C30	1614
BS	8	16
BUFEND	2C8D	1630
BUFFER	2C4D	1629
BUMP	305	517
BUMP0	304	516
BX_REG	2C37	1618
CAPKEY	6	262
CAPS	2C0D	1606
CHAR	2C10	1609
CLRLIN	23F	370
CLRLN1	24B	375
CMDREG	F3	35

Abb. 9.6: Ein Beispiel für eine vom Assembler erzeugte Symboltabelle

Fehlermeldungen

Während des Assemblierungsprozesses stellt der Assembler möglicherweise syntaktische Fehler fest. Er wird sie als einen Teil des abschließenden Ausdrucks aufnehmen. Typische Fehlermeldungen beeinhalten: undefinierte Symbole, bereits definierte Marken, unzulässige Operationscodes, unzulässige Adressen und unzulässige Adressierungsart. Viele zusätzliche Fehlermeldungen sind wünschenswert und üblicherweise vorgesehen. Solche Einrichtungen variieren mit jedem Assembler. Abbildung 9.7 zeigt einen Ausdruck einiger vom Assembler erzeugter Fehlermeldungen.

DIE ASSEMBLERSPRACHE

Wir haben oben Operationscodes behandelt. Nun wollen wir Symbole, Konstanten und Operatoren definieren, die als Teil der Assemblersyntax verwendet werden können.

```
                    1 ;
                    2 ; EIN PROGRAMM ZUR PROGRAMMIERUNG DES 8253
                    3 ; ALS EIN PROGRAMMIERBARES MONOFLOP
                    4 ;
                    5 CNTR1   EQU 31H          ;ADRESSE DES ANSCHLUSSES ZÄHLER 1
                    6 CONP    EQU 33H          ;ADRESSE DES KONTROLLANSCHLUSSES
                    7 ;
0000 B072           8         MOV AL,*72H      ;KONTROLLWORT NACH AL
0002 E633           9         OUT CONP,BL      ;KONTROLLWORT ZUM 8253
*****           INVALID OPERAND *****
                   10 ;
                   11 ; CTR 1,RL = 3,M = 1, BINÄRES ZÄHLEN
                   12 ;
0004 B04B          13          MOV AL,*75      ;DEZIMALE 75
0006 E631          14          OUT CNTR1,AL;  NIEDERWERTIGES BYTE ZUM ZÄHLER 1
                   15          MVI AL,* 00
*****          INVALID MNEMONIC *****
0008 E631          16            OUT (CNTR1)+1,AL;HÖHERWERTIGES BYTE ZUM ZÄHLER 1
*****          INVALID  EXPRESSION  *****
                   17 ;
                   18 ; DER BAUSTEIN IST NUN EINGESTELLT ZUM WARTEN AUF
                   19 ; EINEN AUSLÖSEIMPULS ALS START FÜR DEN ZÄHLER
                   20 ;
                   21 ;
```

Abb. 9.7: Eine Assemblerliste mit Fehlermeldungen

Symbole

Symbole werden zur Darstellung numerischer Werte, entweder Daten oder Adressen, verwendet. Die zulässige Länge der Symbole kann variieren. Üblicherweise fängt ein Symbol mit einem Buchstaben an. Mnemonische Zeichen für den 8086/8088 dürfen nicht als Symbole verwendet werden. Zum Beispiel dürfen Sie kein Symbol wählen, daß mit DX bezeichnet ist, da dies Verwechslungen zwischen dem Symbol und dem DX-Register verursachen würde. Die Dokumentation eines Assemblers gibt an, welche Regeln bei der Definition von Symbolen einzuhalten sind.

Es gibt eine spezielle Assembleranweisung, EQU oder equate (gleichsetzen), die benutzt werden kann, um einem Symbol einen Wert zuzuweisen. Diese Anweisung dient nur als Befehl für den Assembler und wird nicht in Objektcode übersetzt. Abbildung 9.8 gibt ein Beispiel für Symbole, die beim Programmieren verwendet werden, und Werte, die durch die EQU-Anweisung zugewiesen werden.

```
                    1;
                    2;EIN PROGRAMM ZUR PROGRAMMIERUNG DES 8253
                    3;ALS EIN PROGRAMMIERBARES MONOFLOP
                    4;
                    5 CNTR1   EQU 31H      ;ADRESSE DES ANSCHLUSSES ZÄHLER 1
                    6 CONP    EQU 33H      ;ADRESSE DES KONTROLLANSCHLUSSES
                    7;
0000 B72            8         MOV AL,*72H  ;KONTROLLWORT NACH AL
0002 E633           9         OUT CONP,AL  ;KONTROLLWORT ZUM 8253
                   10;
                   11;CTR 1,RL = 3,M = 1, BINÄRES ZÄHLEN
                   12;
0004 B04B          13          MOV AL,* 75  ;DEZIMALE 75
0006 E631          14          OUT CNTR1,AL; NIEDERWERTIGES BYTE ZUM ZÄHLER 1
0008 B000          15          MOV AL,*00
000A E631          16          OUT CNTR1,AL; HÖHERWERTIGES BYTE ZUM ZÄHLER 1
                   17;
                   18;DER BAUSTEIN IST NUN EINGESTELLT ZUM WARTEN AUF
                   19;EINEN AUSLÖSEIMPULS ALS START FÜR DEN ZÄHLER
                   20;
                   21;
```

Abb. 9.8: Beispiele für Symbole, denen Werte durch die EQU-Direktive zugewiesen werden.

Marken

Marken sind spezielle Symbole mit Werten, die nicht vom Programmierer definiert werden müssen. Diese Werte werden automatisch vom Assem-

bler zur Zeit der Programmassemblierung definiert. (Und zwar wird die Adresse der Speicherposition zugewiesen, die der Zeile des Auftretens der Marke entspricht.) Abbildung 9.9 zeigt eine Marke, der vom Assembler ein Wert zugeordnet ist. Wenn das Programm assembliert ist, wird die Speicheradresse von ADDR1 für den MOV-Befehl verwendet.

```
                    1;
                    2;EIN PROGRAMM ZUR PROGRAMMIERUNG DES 8253
                    3;ALS EIN PROGRAMMIERBARES MONOFLOP
                    4;
                    5 CNTR1   EQU 31H          ;ADRESSE DES ANSCHLUSSES ZÄHLER 1
                    6 CONP    EQU 33H          ;ADRESSE DES KONTROLLANSCHLUSSES
                    7;
0000 B072           8         MOV AL,=72H      ;KONTROLLWORT NACH AL
0002 E633           9         OUT CONP,AL      ;KONTROLLWORT ZUM 8253
                   10;
                   11; CTR 1,RL = 3,M = 1, BINÄRES ZÄHLEN
                   12;
0004 B04B          13         MOV AL,=75       ;DEZIMALE 75
0006 E631          14         OUT CNTR1,AL;    NIEDERWERTIGES BYTE ZUM ZÄHLER 1
0008 B000          15         MOV AL,= 00
000A E631          16         OUT CNTR1,AL;    HÖHERWERTIGES BYTE ZUM ZÄHLER 1
000C A0 1000       17         MOV AL. ADDR1    ;LADE DATEN AN ADDR1 NACH AL

000F C3            18         RET              ;RÜCKSPRUNG AUS UNTERPROGRAMM
                   19;
                   20;ES FOLGEN SPEICHERPOSITIONEN FÜR DATEN
                   21;
0010 01            22 ADDR1   DB 01
```

Abb. 9.9: ADDR1 wird von dem MOV-Befehl als eine Adresse benutzt

```
                    1;
                    2;VERSCHIEDENE FORMEN, DATEN MIT EINEM
                    3;8086/8088 ASSEMBLER DARZUSTELLEN
                    4;
0000 B031           5       MOV AL,#00110001P  ;BINÄRE DARSTELLUNG
0002 B031           6       MOV AL,#31H        1HEXADEZIMALE DARSTELLUNG
0004 B031           7       MOV AL,#49         1DEZIMALE DARSTELLUNG
0006 B031           8       MOV AL,#61Q        1OKTALE DARSTELLUNG
                    9;
                   10;
```

Abb. 9.10: Assembler erlauben verschiedene Formen der Datendarstellung

Konstanten oder Literale

Konstanten können dezimal, hexadezimal, oktal, binär oder als alphanumerische Zeichenfolgen ausgedrückt werden. Die Symbole zur Darstellung der Basen der Zahlen variieren von Assembler zu Assembler. Abbildung 9.10 zeigt die verschiedenen Arten, Daten mit einem 8086/8088-Assembler darzustellen.

Operatoren

Um das Schreiben von Programmen in Assemblersprache zu erleichtern, erlaubt der Assembler die Benutzung von *Operatoren*. Mindestens läßt ein Assembler den Gebrauch von Plus und Minus zu; zum Beispiel:

```
MOV   AX, ADR1 + 2
MOV   AX, ADR1 −5
```

Es ist wichtig, zu verstehen, daß der Ausdruck ADR1 + 2 oder ADR1 − 5 vom Assembler zur Zeit der Assemblierung berechnet wird. Abbildung 9.11 zeigt ein Beispiel für die Benutzung eines Operators.

```
                    1;
                    2;  EIN PROGRAMM ZUR DEMONSTRATION DER WIRKUNGSWEISE
                    3;  DER OPERATOREN + UND −
                    4;
                    5 ADDR1   EQU 0123H
                    6;
0000 8B1E2301       7         MOV BX,ADDR1      ;DATEN IN ADDR1 WERDEN IN
                    8                           ;DAS REGISTER BX GEBRACHT
0004 8B0E 2501      9         MOV CX,ADDR1+2    ;DATEN IN ADDR1+2 WERDEN IN
                   10                           ;DAS REGISTER CX GEBRACHT
                   11;
```

Abb. 9.11: Die meisten Assembler erlauben den Gebrauch von Ausdrücken, die zur Zeit der Assemblierung ausgewertet werden

Assemblerdirektiven

Direktiven sind spezielle Kommandos, die der Programmierer dem Assembler gibt. Direktiven werden nicht in Objektcode für den Mikroprozessor übersetzt. Lassen Sie uns einen Überblick über einige der gebräuchlichsten Assemblerdirektiven geben. Wir wollen beginnen mit:

```
ORG     nn
```

Diese Anweisung setzt den Assembleradreßzähler auf den Wert nn. Mit anderen Worten: Der erste ausführbare Befehl nach der ORG-Anweisung beginnt an der Speicherzelle nn. Sie können die ORG- Anweisung an jeder Stelle des Programms benutzen, um den Wert der Assemblerspeicheradresse zu ändern. Eine zweite Assemblerdirektive ist EQU:

```
MY_ADD  EQU  nn
```

Die Marke MY_ADD wird zur Zeit der Assemblierung dem numerischen Wert nn gleichgesetzt.

Um ein einzelnes Byte im Speicher abzulegen, wird die Anweisung DB (define byte, definiere Byte) benutzt, zum Beispiel:

```
ONE  DB  01
```

Der Marke ONE wird der Wert des laufenden Assembleradreßzählerstandes zugewiesen; an dieser Adresse wird die 8-Bit-Größe 00000001 gespeichert. Wenn wir eine Marke als Zeiger auf eine 16-Bit-Größe definieren möchten, können wir die Direktive DW (define word, definiere Wort) benutzen. Sie sieht so aus:

```
TWO  DW  02
```

Die Marke TWO ist gleich der Adresse des 16-Bit-Werts 0000 0000 0000 0010.

Wir können auch einen Zeiger auf ein Doppelwort (oder 32 Bits) definieren, indem wir die Direktive DD (define double word, definiere Doppelwort) verwenden, und zwar in der folgenden Art:

```
BIG_NUMBER  DD  45860912H
```

Der binäre Wert, auf den BIG_NUMBER zeigt, benötigt 4 Bytes im Speicher.

Wenn wir eine Konstante im Speicher für eine Zeichenfolge, die als Meldung gedruckt werden kann, definieren wollen, können wir auch die Direktive DB benutzen. Nehmen wir zum Beispiel an, wir wollten die Meldung „Fehler 5 – Unzulässiger Wert" speichern. Wir könnten sie folgendermaßen in ein Programm einfügen:

```
ERR5  DB  'Fehler 5 – Unzulässiger Wert'
```

Der Assembler schreibt das ASCII-Äquivalent jedes Zeichens in den Speicher. Für die Speicherung dieser Meldung werden 28 Speicherbytes benötigt. Abhängig von der Arbeitsweise des Systems wird möglicherweise ein weiteres Byte verwendet, um die Meldung zu beenden oder ihr

Ende anzuzeigen. Einige Assembler benutzen dafür andere Anweisungen. Geläufige Anweisungssymbole sind DS (define storage, definiere Speicherinhalt) und DT (define text, definiere Text).

Die END-Direktive wird benutzt, um den Assembler zu informieren, daß keine weiteren Befehle zu assemblieren sind. Der Assembler wird dann nach dieser Anweisung keine weiteren Befehle suchen.

DIREKTIVEN ZUR VERWALTUNG DES SEGMENTIERTEN SPEICHERS

Zu Beginn des Kapitels 4 wurde bereits vermerkt, daß die für den 8086/8088 vertriebenen Assembler in einzelnen Punkten voneinander abweichen. Dies betrifft im wesentlichen die verfügbaren Direktiven, die von Fall zu Fall in unterschiedlicher Anzahl und Leistungsfähigkeit und auch unter verschiedenen Namen verfügbar sind. Es ist jedoch nicht Aufgabe dieses Buches, darüber eine Übersicht zu geben. Für die Verwaltung der Segmentregister ist jedoch das Verständnis bestimmter Direktiven erforderlich, die hier kurz beschrieben und an Beispielen erläutert werden. Auf alle Einzelheiten kann jedoch nicht eingegangen werden. Die beschriebenen Direktiven sind kompatibel mit dem Makroassembler für den IBM PC.

Zur Laufzeit eines 8086/8088-Programms liegt jeder Befehl und jeder Datenwert innerhalb eines bestimmten Speichersegments. Wie aus dem 16-Bit-Offset eines Operanden innerhalb eines Segments die physikalische 20-Bit-Adresse erzeugt wird, haben wir bereits im Kapitel 3 kennengelernt. Anwendungen, bei denen Programmcode und Daten mehr als 64 KByte umfassen, können nicht mehr innerhalb eines Segments untergebracht werden. Dem Assembler muß deswegen mitgeteilt werden, wie das zu übersetzende Programm in Segmente aufgeteilt werden soll. Dieses Zweck erfüllt die SEGMENT-Direktive. Mit dieser Direktive kann der Programmierer einen Programmabschnitt als ein Segment definieren und diesem Segment u.a. einen Namen geben. Ein Segment wird durch die Anweisung ENDS abgeschlossen.

Beispiel:

```
SORTIEREN  SEGMENT ; Z. B. PROZEDUREN FÜR
                   ; SORTIERVERFAHREN
           .
           .
SORTIEREN  ENDS
DATEN      SEGMENT ; SATZ ZU SORTIERENDER
                   ; DATEN
           .
           .
DATEN      ENDS
```

Der Name eines Segments kann als Operand in einem MOV-Befehl verwendet werden:

```
MOV     AX,DATEN ; LADE DS MIT ZEIGER
MOV     DS,AX    ; AUF DATENSEGMENT
```

oder in einer Assume-Direktive angegeben werden:

```
ASSUME DS:DATEN,CS:SORTIEREN
```

Diese Direktive teilt dem Assembler mit, welche Werte das Datensegmentregister und das Codesegmentregister für das folgende Programmstück haben sollen. Diese Information nutzt der Assembler dann zur Überprüfung der Adressierbarkeit der Operanden und gibt sie zusätzlich an das Binder-/Laderprogramm weiter. So wird zum Beispiel vom Assembler eine Fehlermeldung erzeugt, wenn in einem MOV-Befehl ein Operand adressiert wird, der nicht in dem durch die letzte ASSUME-Direktive angegebenen Datensegment liegt.

Marken für Sprungziele, die der Anfang einer Prozedur sind, können durch die PROC-Direktive gekennzeichnet werden. Das Ende der Prozedur wird durch ENDP angegeben. Die Prozedur kann zusätzlich über die Typattribute NEAR und FAR gekennzeichnet werden. Im Falle FAR werden sämtliche RET-Befehle innerhalb der Prozedur als Intersegment-Rücksprung, jeder Aufruf der Prozedur als Intersegment-Aufruf erzeugt. Das folgende Beispiel zeigt einen Weg, wie eine Prozedur über zwei verschiedene Namen sowohl über einen Intersegment- als auch über einen Intrasegment-CALL-Befehl aufgerufen werden kann. Aufrufe innerhalb des eigenen Codesegments können so durch den schnelleren, weil weniger aufwendigen Intrasegment-Aufruf erfolgen, und die Prozedur kann trotzdem von anderen Segmenten erreicht werden.

Beispiel:

```
CSEG1       SEGMENT
INTRA_PROC  PROC  NEAR
                .
                .
            ENDP
INTER_PROC  PROC  FAR
            CALL  INTRA_PROC
            RET
            ENDP
CSEG1       ENDS
```

```
CSEG2       SEGMENT
              .
            CALL INTER_PROC ; ERLAUBT
            CALL INTRA_PROC ; IN CSEG2 NICHT
                            ; ERLAUBT!!
              .
CSEG2       ENDP
```

ZUSAMMENFASSUNG

In diesem Kapitel haben wir die zur Programmentwicklung erforderlichen Techniken und Hardware- sowie Softwarewerkzeuge untersucht. Wir haben auch die verschiedenen Alternativen unter den zur Programmentwicklung zur Verfügung stehenden Werkzeugen behandelt. Die behandelten Techniken reichten von Einplatinen-Mikrocomputern bis zum vollständigen Entwicklungssystem. Mit einer Untersuchung einiger allgemeingültiger Punkte über Mikroprozessorassemblerprogramme haben wir die Diskussion abgeschlossen.

SCHLUSSBEMERKUNG

In diesem Buch haben wir die wichtigen Aspekte der Programmierung des 8086/8088 vollständig behandelt. Das erste Kapitel begann mit einer Diskussion der grundlegenden Definitionen und Konzepte der Programmierung. Dann fuhren wir mit der Erforschung und Erörterung der Besonderheiten der 8086/8088-CPU fort. Als nächstes untersuchten wir den Gebrauch der E/A-Einrichtungen und der verschiedenen Werkzeuge der Softwareentwicklung.

Was ist nun der nächste Schritt? Nachdem Sie dieses Buch gelesen haben, sollten Sie soweit sein, allein weiterzuarbeiten und Ihre eigenen Programme auf dem 8086/8088 zu schreiben. Wenn Sie damit fortfahren, werden Sie entdecken, daß der Befehlssatz für diese Mikroprozessoren sehr leistungsfähig ist. Während Sie mehr und mehr Erfahrung im Gebrauch der verschiedenen Befehle gewinnen, werden Sie neue und kreative Wege zur Lösung Ihrer Anwendungsprobleme entdecken. Dort beginnt der richtige Spaß.

Und nun viel Glück und erfolgreiches Programmieren!

Anhang A

Umwandlungstabellen Hexadezimal – Dezimal

HEX	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F	00	000
0	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	0	0
1	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31	256	4096
2	32	33	34	35	36	37	38	39	40	41	42	43	44	45	46	47	512	8192
3	48	49	50	51	52	53	54	55	56	57	58	59	60	61	62	63	768	12288
4	64	65	66	67	68	69	70	71	72	73	74	75	76	77	78	79	1024	16384
5	80	81	82	83	84	85	86	87	88	89	90	91	92	93	94	95	1280	20480
6	96	97	98	99	100	101	102	103	104	105	106	107	108	109	110	111	1536	24576
7	112	113	114	115	116	117	118	119	120	121	122	123	124	125	126	127	1792	28672
8	128	129	130	131	132	133	134	135	136	137	138	139	140	141	142	143	2048	32768
9	144	145	146	147	148	149	150	151	152	153	154	155	156	157	158	159	2304	36864
A	160	161	162	163	164	165	166	167	168	169	170	171	172	173	174	175	2560	40960
B	176	177	178	179	180	181	182	183	184	185	186	187	188	189	190	191	2816	45056
C	192	193	194	195	196	197	198	199	200	201	202	203	204	205	206	207	3072	49152
D	208	209	210	211	212	213	214	215	216	217	218	219	220	221	222	223	3328	53248
E	224	225	226	227	228	229	230	231	232	233	234	235	236	237	238	239	3584	57344
F	240	241	242	243	244	245	246	247	248	249	250	251	252	253	254	255	3840	61440

5		4		3		2		1		0	
HEX	DEZ	HEX	DEZ	HEX	DEZ	HEX	DEZ	HEX	DEZ	HEX	DEZ
0	0	0	0	0	0	0	0	0	0	0	0
1	1,048,576	1	65,536	1	4,096	1	256	1	16	1	1
2	2,097,152	2	131,072	2	8,192	2	512	2	32	2	2
3	3,145,728	3	196,608	3	12,288	3	768	3	48	3	3
4	4,194,304	4	262,144	4	16,384	4	1,024	4	64	4	4
5	5,242,880	5	327,680	5	20,480	5	1,280	5	80	5	5
6	6,291,456	6	393,216	6	24,576	6	1,536	6	96	6	6
7	7,340,032	7	458,752	7	28,672	7	1,792	7	112	7	7
8	8,388,608	8	524,288	8	32,768	8	2,048	8	128	8	8
9	9,437,184	9	589,824	9	36,864	9	2,304	9	144	9	9
A	10,485,760	A	655,360	A	40,960	A	2,560	A	160	A	10
B	11,534,336	B	720,896	B	45,056	B	2,816	B	176	B	11
C	12,582,912	C	786,432	C	49,152	C	3,072	C	192	C	12
D	13,631,488	D	851,968	D	53,248	D	3,328	D	208	D	13
E	14,680,064	E	917,504	E	57,344	E	3,584	E	224	E	14
F	15,728,640	F	983,040	F	61,440	F	3,840	F	240	F	15

Anhang B

ASCII-Tabelle

HEX	MSD	0	1	2	3	4	5	6	7
LSD	BITS	000	001	010	011	100	101	110	111
0	0000	NUL	DLE	SPACE	0	@	P	—	p
1	0001	SOH	DC1	!	1	A	Q	a	q
2	0010	STX	DC2	"	2	B	R	b	r
3	0011	ETX	DC3	#	3	C	S	c	s
4	0100	EOT	DC4	$	4	D	T	d	t
5	0101	ENQ	NAK	%	5	E	U	e	u
6	0110	ACK	SYN	&	6	F	V	f	v
7	0111	BEL	ETB	'	7	G	W	g	w
8	1000	BS	CAN	(	8	H	X	h	x
9	1001	HT	EM	)	9	I	Y	i	y
A	1010	LF	SUB	*	:	J	Z	j	z
B	1011	VT	ESC	+	;	K	[	k	{
C	1100	FF	FS	,	<	L	\	l	|
D	1101	CR	GS	–	=	M	]	m	}
E	1110	SO	RS	.	>	N	^	n	~
F	1111	SI	US	/	?	O	←	o	DEL

DIE ASCII SYMBOLE

NUL Null
SOH Beginn des Kopfes (Start of Heading)
STX Beginn des Textes (Start of Text)
ETX Ende des Textes (End of Text)
EOT Ende der Übertragung (End of Transmission)
ENQ Anfrage (Enquing)
ACK Bestätigung (Acknowledge)
BEL Klingel (Bell)
BS Zurücksetzen (Backspace)
HT Horizontaler Tabulator (Horizontal Tabulation)
LF Zeilenvorschub (Line Feed)
VT Vertikaler Tabulator (Vertical Tabulation)
FF Format Vorschub (Form Feed)
CR Wagenrücklauf/Zeilenwechsel (Carriage Return)
SO Rückschaltung (Shift Out)
SI Dauerumschaltung (Shift In)
DLE Datenverbindungs-Umschaltung (Data Link Escape)
DC Geratesteuerung (Device Control)
NAK Negativ-Bestätigung (Negative Acknowledge)
SYN Synchronisations-Leerlauf (Synchronous Idle)
ETB Ende des Übertragungsblockes (End of Transmission Block)
CAN Annullieren (Cancel)
EM Datenträgerende (End of Medium)
SUB Ersetzen (Substitute)
ESC Umschaltung (Escape)
FS Dateitrennzeichen (File Separator)
GS Gruppentrennzeichen (Group Separator)
RS Satztrennzeichen (Record Separator)
US Einheiten-Trennzeichen (Unit Separator)
SP Leerzeichen (Space/Blank)
DEL Löschzeichen (Delete)

Anhang C

Umwandlungstabelle BCD – DEZIMAL

DEZIMAL	BCD	DEZIMAL	BCD	DEZIMAL	BCD
0	0000	10	00010000	91	10010000
1	0001	11	00010001	91	10010001
2	0010	12	00010010	92	10010010
3	0011	13	00010011	93	10010011
4	0100	14	00010100	94	10010100
5	0101	15	00010101	95	10010101
6	0110	16	00010110	96	10010110
7	0111	17	00010111	97	10010111
8	1000	18	00011000	98	10011000
9	1001	19	00011001	99	10011001

Anhang D

Referenzdaten zum Befehlssatz des 8086/8088

AAA	**AAA** keine Operanden ASCII - Anpassung für Addition			**Flags**	O D I T S Z A P C U U U X U X
Operanden	**Benötigte Taktzyklen**	**Anzahl der Datenüber-tragungen***	**Bytes**	**Beispiel**	
keine Operanden	4	–	1	AAA	

AAD	**AAD** keine Operanden ASCII - Anpassung für Division			**Flags**	O D I T S Z A P C U X X U X U
Operanden	**Benötigte Taktzyklen**	**Anzahl der Datenüber-tragungen***	**Bytes**	**Beispiel**	
keine Operanden	60	–	2	AAD	

AAM	**AAM** keine Operanden ASCII - Anpassung für Multiplikation			**Flags**	O D I T S Z A P C U X X U X U
Operanden	**Benötigte Taktzyklen**	**Anzahl der Datenüber-tragungen***	**Bytes**	**Beispiel**	
keine Operanden	83	–	1	AAM	

AAS	**AAS** keine Operanden ASCII - Anpassung für Subtraktion			**Flags**	O D I T S Z A P C U U U X U X
Operanden	**Benötigte Taktzyklen**	**Anzahl der Datenüber-tragungen***	**Bytes**	**Beispiel**	
keine Operanden	4	–	1	AAS	

ADC	**ADC** Ziel, Quelle ASCII mit Übertrag addieren			**Flags**	O D I T S Z A P C X X X X X
Operanden	**Benötigte Taktzyklen**	**Anzahl der Datenüber-tragungen***	**Bytes**	**Beispiel**	
Register, Register	3	–	2	ADC AX, SI	
Register, Speicher	9+EA	1	2-4	ADC DX, BETA [SI]	
Speicher, Register	16+EA	2	2-4	ADC ALPHA [BX] [SI], DI	
Register, unmittelbar	4	–	3-4	ADC BX, 256	
Speicher, unmittelbar	17+EA	2	3-6	ADC GAMMA, 30H	
Akkumulator, unmittelbar	4	–	2-3	ADC AL, 5	

* Addieren Sie beim 8086 vier Taktzyklen für jede Übergabe eines 16-Bit-Wortes, das an einer ungeraden Adresse beginnt. Beim 8088 müssen Sie für jede Übertragung eines 16-Bit-Wortes vier Taktzyklen zusätzlich einkalkulieren.

Mnemonics © Intel, 1978

ADD	**ADD** Ziel, Quelle Addieren			**Flags**	O D I T S Z A P C X X X X X
Operanden	**Benötigte Taktzyklen**	**Anzahl der Datenübertragungen***	**Bytes**	**Beispiel**	
Register, Register	3	–	2	ADD CX, DX	
Register, Speicher	9+EA	1	2-4	ADD DI, [BX].ALPHA	
Speicher, Register	16+EA	2	2-4	ADD TEMP, CL	
Register, unmittelbar	4	–	3-4	ADD CL, 2	
Speicher, unmittelbar	17+EA	2	3-6	ADD ALPHA, 2	
Akkumulator, unmittelbar	4	–	2-3	ADD AX, 200	

AND	**AND** Ziel, Quelle Logisches UND bilden			**Flags**	O D I T S Z A P C 0 X X U X 0
Operanden	**Benötigte Taktzyklen**	**Anzahl der Datenübertragungen***	**Bytes**	**Beispiel**	
Register, Register	3	–	2	AND AL, BL	
Register, Speicher	9+EA	1	2-4	AND CX,FLAG_WORD	
Speicher, Register	16+EA	2	2-4	AND ASCII [DI],AL	
Register, unmittelbar	4	–	3-4	AND CX,0F0H	
Speicher, unmittelbar	17+EA	2	3-6	AND BETA, 01H	
Akkumulator, unmittelbar	4	–	2-3	AND AX, 01010000 B	

CALL	**CALL** Prozedurname Prozedur aufrufen			**Flags**	O D I T S Z A P C
Operanden	**Benötigte Taktzyklen**	**Anzahl der Datenübertragungen***	**Bytes**	**Beispiel**	
NEAR-Prozedur	19	1	3	CALL NEAR_PROC	
FAR-Prozedur	28	2	5	CALL FAR_PROC	
Speicherzeiger (16 bit)	21+EA	2	2-4	CALL PROC_TABLE [SI]	
Registerzeiger (16 bit)	16	1	2	CALL AX	
Speicherzeiger (32 bit)	37+EA	4	2-4	CALL [BX].TASK [SI]	

CBW	**CBW** Keine Operanden Byte in Wort umwandeln			**Flags**	O D I T S Z A P C
Operanden	**Benötigte Taktzyklen**	**Anzahl der Datenübertragungen***	**Bytes**	**Beispiel**	
Keine Operanden	2	–	1	CBW	

CLC	**CLC** Keine Operanden Übertragskennzeichen löschen			**Flags**	O D I T S Z A P C 0
Operanden	**Benötigte Taktzyklen**	**Anzahl der Datenübertragungen***	**Bytes**	**Beispiel**	
Keine Operanden	2	–	1	CLC	

CLD	**CLD** Keine Operanden Richtungskennzeichen löschen			**Flags**	O D I T S Z A P C 0
Operanden	**Benötigte Taktzyklen**	**Anzahl der Datenübertragungen***	**Bytes**	**Beispiel**	
Keine Operanden	2	–	1	CLD	

* Addieren Sie beim 8086 vier Taktzyklen für jede Übergabe eines 16-Bit-Wortes, das an einer ungeraden Adresse beginnt. Beim 8088 müssen Sie für jede Übertragung eines 16-Bit-Wortes vier Taktzyklen zusätzlich einkalkulieren.

Mnemonics © Intel, 1978

CLI	**CLI** Keine Operanden Unterbrechungsfreigabekennzeichen loschen			**Flags** O D I T S Z A P C 0
Operanden	**Benötigte Taktzyklen**	**Anzahl der Datenüber-tragungen***	**Bytes**	**Beispiel**
Keine Operanden	2	–	1	CLI

CMC	**CMC** Keine Operanden Übertragskennzeichen komplementieren			**Flags** O D I T S Z A P C X
Operanden	**Benötigte Taktzyklen**	**Anzahl der Datenüber-tragungen***	**Bytes**	**Beispiel**
Keine Operanden	2	–	1	CMC

CMP	**CMP** Ziel, Quelle Ziel mit Quelle vergleichen			**Flags** O D I T S Z A P C X X X X X
Operanden	**Benötigte Taktzyklen**	**Anzahl der Datenüber-tragungen***	**Bytes**	**Beispiel**
Register, Register	3	–	2	CMP BX, CX
Register, Speicher	9+EA	1	2-4	CMP DH, ALPHA
Speicher, Register	9+EA	1	2-4	CMP [BP+2], SI
Register, unmittelbar	4	–	3-4	CMP BL, 02H
Speicher, unmittelbar	10+EA	1	3-6	CMP [BX].RADAR [DI],3420H
Akkumulator, unmittelbar	4	–	2-3	CMP AL, 00010000B

CMPS	**CMPS** Ziel-String, Quell-String Zeichenketten vergleichen			**Flags** O D I T S Z A P C X X X X X
Operanden	**Benötigte Taktzyklen**	**Anzahl der Datenüber-tragungen***	**Bytes**	**Beispiel**
Ziel-String, Quell-String	22	2	1	CMPS BUFF1, BUFF2
(wiederhole) Ziel-String, Quell-String	9+22/wdhlg	2/wdhlg	1	REPE CMPS ID, KEY

CWD	**CWD** Keine Operanden Wort zu Doppelwort umwandeln			**Flags** O D I T S Z A P C
Operanden	**Benötigte Taktzyklen**	**Anzahl der Datenüber-tragungen***	**Bytes**	**Beispiel**
Keine Operanden	5	–	1	CWD

DAA	**DAA** Keine Operanden Dezimale Anpassung für Addition			**Flags** O D I T S Z A P C X X X X X
Operanden	**Benötigte Taktzyklen**	**Anzahl der Datenüber-tragungen***	**Bytes**	**Beispiel**
Keine Operanden	4	–	1	DAA

DAS	**DAS** Keine Operanden Dezimale Anpassung für Subtraktion			**Flags** O D I T S Z A P C U X X X X X
Operanden	**Benötigte Taktzyklen**	**Anzahl der Datenüber-tragungen***	**Bytes**	**Beispiel**
Keine Operanden	4	–	1	DAS

* Addieren Sie beim 8086 vier Taktzyklen für jede Übergabe eines 16-Bit-Wortes, das an einer ungeraden Adresse beginnt. Beim 8088 müssen Sie für jede Übertragung eines 16-Bit-Wortes vier Taktzyklen zusätzlich einkalkulieren.

DEC	**DEC** Ziel Dekrementieren			**Flags** O D I T S Z A P C X X X X X
Operanden	**Benötigte Taktzyklen**	**Anzahl der Datenübertragungen***	**Bytes**	**Beispiel**
Register (16 Bit)	2	–	1	DEC AX
Register (8 Bit)	3	–	2	DEC AL
Speicher	15+EA	2	2-4	DEC ARRAY [SI]

DIV	**DIV** Quelle Division (ohne Vorzeichen)			**Flags** O D I T S Z A P C U U U U U
Operanden	**Benötigte Taktzyklen**	**Anzahl der Datenübertragungen***	**Bytes**	**Beispiel**
Register (8 Bit)	80-90	–	2	DIV CL
Register (16 Bit)	144-162	–	2	DIV BX
Speicher (8 Bit)	86-96 +EA	1	2-4	DIV ALPHA
Speicher (16 Bit)	(150-168) +EA	1	2-4	DIV TABLE [s]

ESC	**ESC** Externer Opcode, Quelle Objektcode und Operand für Koprozessor bereitstellen			**Flags** O D I T S Z A P C
Operanden	**Benötigte Taktzyklen**	**Anzahl der Datenübertragungen***	**Bytes**	**Beispiel**
unmittelbar, Speicher	8+EA	1	2-4	ESC 6,ARRAYX [SI]
unmittelbar, Register	2	–	2	ESC 20,AL

HLT	**HLT** Keine Operanden Halten			**Flags** O D I T S Z A P C
Operanden	**Benötigte Taktzyklen**	**Anzahl der Datenübertragungen***	**Bytes**	**Beispiel**
Keine Operanden	2	–	1	HLT

IDIV	**IDIV** Quelle ganzzahlig dividieren			**Flags** O D I T S Z A P C U U U U U
Operanden	**Benötigte Taktzyklen**	**Anzahl der Datenübertragungen***	**Bytes**	**Beispiel**
Register (8 Bit)	101-112	–	2	IDIV BL
Register (16 Bit)	165-184	–	2	IDIV CX
Speicher (8 Bit)	(107-118) +EA	1	2-4	IDIV DIVISOR_BYTE [SI]
Speicher (16 Bit)	(171-190) +EA	1	2-4	IDIV [BX].DIVISOR_WORD

* Addieren Sie beim 8086 vier Taktzyklen für jede Übergabe eines 16-Bit-Wortes, das an einer ungeraden Adresse beginnt. Beim 8088 müssen Sie für jede Übertragung eines 16-Bit-Wortes vier Taktzyklen zusätzlich einkalkulieren.
Mnemonics © Intel, 1978

IMUL	IMUL Quelle ganzzahlig multiplizieren			Flags O D I T S Z A P C X U U U U X
Operanden	**Benötigte Taktzyklen**	**Anzahl der Datenübertragungen***	**Bytes**	**Beispiel**
Register (8 Bit)	80-98	–	2	IMUL CL
Register (16 Bit)	128-154	–	2	IMUL BX
Speicher (8 Bit)	(86-104) + EA	1	2-4	IMUL RATE BYTE
Speicher (16 Bit)	(134-160) + EA	1	2-4	IMUL RATE WORD [BP] [DI]

IN	IN Akkumulator, Anschluß Byte od. Wort von E/A-Anschluß einlesen			Flags O D I T S Z A P C
Operanden	**Benötigte Taktzyklen**	**Anzahl der Datenübertragungen***	**Bytes**	**Beispiel**
Akkumulator, unmittelbar (8 Bit)	10	1	2	IN AL, 0FFEAH
Akkumulator, DX	8	1	1	IN AX, DX

INC	INC Ziel Inkrementieren			Flags O D I T S Z A P C X X X X
Operanden	**Benötigte Taktzyklen**	**Anzahl der Datenübertragungen***	**Bytes**	**Beispiel**
Register (16 Bit)	2	–	1	INC CX
Register (8 Bit)	3	–	2	INC BL
Speicher	15 + EA	2	2-4	INC ALPHA [DI] [BX]

INT	INT Unterbrechungstyp (0-255) Unterbrechen			Flags O D I T S Z A P C 0 0
Operanden	**Benötigte Taktzyklen**	**Anzahl der Datenübertragungen***	**Bytes**	**Beispiel**
unmittelbar (8 Bit, Typ = 3)	52	5	1	INT 3
unmittelbar (8 Bit, Typ ≠ 3)	51	5	2	INT 67

INTR †	INTR externe maskierbare Unterbrechung Unterbrechung wenn INTR und IF = 1			Flags O D I T S Z A P C 0 0
Operanden	**Benötigte Taktzyklen**	**Anzahl der Datenübertragungen***	**Bytes**	**Beispiel**
Keine Operanden	61	7	N/A	N/A

INTO	INTO Keine Operanden Bei gesetztem Überlaufkennzeichen unterbrechen			Flags O D I T S Z A P C 0 0
Operanden	**Benötigte Taktzyklen**	**Anzahl der Datenübertragungen***	**Bytes**	**Beispiel**
Keine Operanden	53 oder 4	5	1	INTO

* Addieren Sie beim 8086 vier Taktzyklen für jede Übergabe eines 16-Bit-Wortes, das an einer ungeraden Adresse beginnt. Beim 8088 müssen Sie für jede Übertragung eines 16-Bit-Wortes vier Taktzyklen zusätzlich einkalkulieren.
† INTR ist kein Befehl; es dient hier nur zur Information über Zeitabläufe.
Mnemonics ©Intel, 1978

IRET	IRET Keine Operanden Rückkehr von Unterbrechung			Flags O D I T S Z A P C R R R R R R R R R
Operanden	**Benötigte Taktzyklen**	**Anzahl der Datenübertragungen***	**Bytes**	**Beispiel**
Keine Operanden	24	3	1	IRET

JA/JNBE	JA/JNBE Kurz-Marke / Sprung bei vorzeichenlos größer / Sprung bei vorzeichenlos nicht kleiner od. gleich			Flags O D I T S Z A P C
Operanden	**Benötigte Taktzyklen**	**Anzahl der Datenübertragungen***	**Bytes**	**Beispiel**
Kurz-Marke	16 oder 4	–	2	JA ABOVE

JAE/JNB	JAE/JNB Kurz-Marke / Sprung bei vorzeichenlos größer od. gleich / Sprung bei vorzeichenlos nicht kleiner			Flags O D I T S Z A P C
Operanden	**Benötigte Taktzyklen**	**Anzahl der Datenübertragungen***	**Bytes**	**Beispiel**
Kurz-Marke	16 oder 4	–	2	JAE ABOVE EQUAL

JB/JNE	JB/JNAE Kurz-Marke Sprung bei vorzeichenlos kleiner / Sprung bei vorzeichenlos nicht größer od. gleich			Flags O D I T S Z A P C
Operanden	**Benötigte Taktzyklen**	**Anzahl der Datenübertragungen***	**Bytes**	**Beispiel**
Kurz-Marke	16 oder 4	–	2	JB BELOW

JBE/JNA	JBE/JNA Kurz-Marke / Sprung bei vorzeichenlos kleiner od. gleich / Sprung bei vorzeichenlos nicht größer			Flags O D I T S Z A P C
Operanden	**Benötigte Taktzyklen**	**Anzahl der Datenübertragungen***	**Bytes**	**Beispiel**
Kurz-Marke	16 oder 4	–	2	JNA NOT ABOVE

JC	JC Kurz-Marke Sprung bei gesetztem Übertragskennzeichen			Flags O D I T S Z A P C
Operanden	**Benötigte Taktzyklen**	**Anzahl der Datenübertragungen***	**Bytes**	**Beispiel**
Kurz-Marke	16 oder 4	–	2	JC CARRY SET

JCXZ	JCXZ Kurz-Marke Sprung, falls CX-Register Null ist			Flags O D I T S Z A P C
Operanden	**Benötigte Taktzyklen**	**Anzahl der Datenübertragungen***	**Bytes**	**Beispiel**
Kurz-Marke	18 oder 4	–	2	JCXZ COUNT DONE

JE/JZ	JE/JZ Kurz-Marke Sprung bei gleich / Sprung bei Null			Flags O D I T S Z A P C
Operanden	**Benötigte Taktzyklen**	**Anzahl der Datenübertragungen***	**Bytes**	**Beispiel**
Kurz-Marke	16 oder 4	–	2	JZ ZERO

* Addieren Sie beim 8086 vier Taktzyklen für jede Übergabe eines 16-Bit-Wortes, das an einer ungeraden Adresse beginnt. Beim 8088 müssen Sie für jede Übertragung eines 16-Bit-Wortes vier Taktzyklen zusätzlich einkalkulieren.

Mnemonics © Intel, 1978

JG/JNLE	JG/JNLE Kurz-Marke Sprung bei größer / Sprung bei nicht kleiner od. gleich			Flags O D I T S Z A P C
Operanden	**Benötigte Taktzyklen**	**Anzahl der Datenüber-tragungen***	**Bytes**	**Beispiel**
Kurz-Marke	16 oder 4	–	2	JG GREATER

JGE/JNL	JGE/JNL Kurz-Marke / Sprung bei größer oder gleich / Sprung bei nicht kleiner			Flags O D I T S Z A P C
Operanden	**Benötigte Taktzyklen**	**Anzahl der Datenüber-tragungen***	**Bytes**	**Beispiel**
Kurz-Marke	16 oder 4	–	2	JGE GREATER_EQUAL

JL/JNGE	JL/JNGE Kurz-Marke / Sprung bei kleiner / Sprung bei nicht größer oder gleich			Flags O D I T S Z A P C
Operanden	**Benötigte Taktzyklen**	**Anzahl der Datenüber-tragungen***	**Bytes**	**Beispiel**
Kurz-Marke	16 oder 4	–	2	JL LESS

JLE/JNG	JLE/JNG Kurz-Marke / Sprung bei kleiner oder gleich / Sprung bei nicht größer			Flags O D I T S Z A P C
Operanden	**Benötigte Taktzyklen**	**Anzahl der Datenüber-tragungen***	**Bytes**	**Beispiel**
Kurz-Marke	16 oder 4	–	2	JNG NOT_GREATER

JMP	JMP Sprungziel unbedingter Sprung			Flags O D I T S Z A P C
Operanden	**Benötigte Taktzyklen**	**Anzahl der Datenüber-tragungen***	**Bytes**	**Beispiel**
Kurz-Marke	15	–	2	JMP SHORT
NEAR-Marke	15	–	3	JMP WITHIN_SEGMENT
FAR-Marke	15	–	5	JMP FAR_LABEL
Speicherzeiger (16 Bit)	18 + EA	1	2-4	JMP [BX].TARGET
Registerzeiger (16 Bit)	11	–	2	JMP CX
Speicherzeiger (32 Bit)	24 + EA	2	2-4	JMP OTHER:SEG [SI]

JNC	JNC Kurz-Marke Sprung bei nicht gesetztem Übertragskennzeichen			Flags O D I T S Z A P C
Operanden	**Benötigte Taktzyklen**	**Anzahl der Datenüber-tragungen***	**Bytes**	**Beispiel**
Kurz-Marke	16 oder 4	–	2	JNC NOT_CARRY

JNE/JNZ	JNE/JNZ Kurz-Marke Sprung bei nicht gleich / Sprung bei nicht Null			Flags O D I T S Z A P C
Operanden	**Benötigte Taktzyklen**	**Anzahl der Datenüber-tragungen***	**Bytes**	**Beispiel**
Kurz-Marke	16 oder 4	–	2	JNE NOT_EQUAL

* Addieren Sie beim 8086 vier Taktzyklen für jede Übergabe eines 16-Bit-Wortes, das an einer ungeraden Adresse beginnt. Beim 8088 müssen Sie für jede Übertragung eines 16-Bit-Wortes vier Taktzyklen zusätzlich einkalkulieren.
Mnemonics © Intel, 1978

JNO	**JNO** Kurz-Marke Sprung bei nicht gesetzten Überlaufskennzeichen			**Flags** O D I T S Z A P C
Operanden	**Benötigte Taktzyklen**	**Anzahl der Datenübertragungen***	**Bytes**	**Beispiel**
Kurz-Marke	16 oder 4	–	2	JNO NO_OVERFLOW

JNP/JPO	**JNP/JPO** Kurz-Marke / Sprung bei nicht gesetztem Paritätszeichen / Sprung bei ungerader Parität			**Flags** O D I T S Z A P C
Operanden	**Benötigte Taktzyklen**	**Anzahl der Datenübertragungen***	**Bytes**	**Beispiel**
Kurz-Marke	16 oder 4	–	2	JPO ODD_PARITY

JNS	**JNS** Kurz-Marke Sprung bei nicht gesetztem Vorzeichenkennzeichen			**Flags** O D I T S Z A P C
Operanden	**Benötigte Taktzyklen**	**Anzahl der Datenübertragungen***	**Bytes**	**Beispiel**
Kurz-Marke	16 oder 4	–	2	JNS POSITIVE

JO	**JO** Kurz-Marke Sprung bei gesetztem Überlaufskennzeichen			**Flags** O D I T S Z A P C
Operanden	**Benötigte Taktzyklen**	**Anzahl der Datenübertragungen***	**Bytes**	**Beispiel**
Kurz-Marke	16 oder 4	–	2	JO SIGNED_OVRFLW

JP/JPE	**JP/JPE** Kurz-Marke / Sprung bei gesetztem Paritätszeichen / Sprung bei gerader Parität			**Flags** O D I T S Z A P C
Operanden	**Benötigte Taktzyklen**	**Anzahl der Datenübertragungen***	**Bytes**	**Beispiel**
Kurz-Marke	16 oder 4	–	2	JPE EVEN_PARITY

JS	**JS** Kurz-Marke Sprung bei gesetztem Vorzeichenkennzeichen			**Flags** O D I T S Z A P C
Operanden	**Benötigte Taktzyklen**	**Anzahl der Datenübertragungen***	**Bytes**	**Beispiel**
Kurz-Marke	16 oder 4	–	2	JS NEGATIVE

LAHF	**LAHF** Keine Operanden Register AH mit Kennzeichenbits laden			**Flags** O D I T S Z A P C
Operanden	**Benötigte Taktzyklen**	**Anzahl der Datenübertragungen***	**Bytes**	**Beispiel**
Keine Operanden	4	–	1	LAHF

LDS	**LDS** Ziel, Quelle Zeiger einschließlich DS laden			**Flags** O D I T S Z A P C
Operanden	**Benötigte Taktzyklen**	**Anzahl der Datenübertragungen***	**Bytes**	**Beispiel**
Register (16 Bit), Speicher (32 Bit)	16 + EA	2	2-4	LDS SI,DATA.SEG [DI]

* Addieren Sie beim 8086 vier Taktzyklen für jede Übergabe eines 16-Bit-Wortes, das an einer ungeraden Adresse beginnt. Beim 8088 müssen Sie für jede Übertragung eines 16-Bit-Wortes vier Taktzyklen zusätzlich einkalkulieren.

Mnemonics ©Intel, 1978

LEA	**LEA** Ziel, Quelle Effektive Adresse laden			**Flags** O D I T S Z A P C
Operanden	**Benötigte Taktzyklen**	**Anzahl der Datenübertragungen***	**Bytes**	**Beispiel**
Register (16 Bit), Speicher (16 Bit)	2+EA	–	2-4	LEA BX, [BP][DI]

LES	**LES** Ziel, Quelle Zeiger einschließlich ES laden			**Flags** O D I T S Z A P C
Operanden	**Benötigte Taktzyklen**	**Anzahl der Datenübertragungen***	**Bytes**	**Beispiel**
Register (16 Bit), Speicher (32 Bit)	16+EA	2	2-4	LES DI, [BX].TEXT_BUFF

LOCK	**LOCK** Keine Operanden LOCK-Signal auf dem Bus durchsetzen			**Flags** O D I T S Z A P C
Operanden	**Benötigte Taktzyklen**	**Anzahl der Datenübertragungen***	**Bytes**	**Beispiel**
Keine Operanden	2	–	1	LOCK XCHG FLAG,AL

LODS	**LODS** Quell-Zeichenkette Zeichenkette laden			**Flags** O D I T S Z A P C
Operanden	**Benötigte Taktzyklen**	**Anzahl der Datenübertragungen***	**Bytes**	**Beispiel**
Quell-Zeichenkette (wiederhole) Quell-Zeichenkette	12 9+13 Wdhlg	1 1/Wdhlg	1 1	LODS CUSTOMER_NAME REP LODS NAME

LOOP	**LOOP** Kurz-Marke Schleife, falls CX nicht Null			**Flags** O D I T S Z A P C
Operanden	**Benötigte Taktzyklen**	**Anzahl der Datenübertragungen***	**Bytes**	**Beispiel**
Kurz-Marke	17/5	–	2	LOOP AGAIN

LOOPE/LOOPZ	**LOOPE/LOOPZ** Kurz-Marke Schleife, solange gleich/Schleife solange Null			**Flags** O D I T S Z A P C
Operanden	**Benötigte Taktzyklen**	**Anzahl der Datenübertragungen***	**Bytes**	**Beispiel**
Kurz-Marke	18 oder 6	–	2	LOOPE AGAIN

LOOPNE/LOOPNZ	**LOOPNE/LOOPNZ** Kurz-Marke Schleife, solange n. gleich/Schleife, solange n. Null			**Flags** O D I T S Z A P C
Operanden	**Benötigte Taktzyklen**	**Anzahl der Datenübertragungen***	**Bytes**	**Beispiel**
Kurz-Marke	19 oder 5	–	2	LOOPNE AGAIN

NMI†	**NMI** externe nicht maskierbare Unterbrechung Unterbrechung wenn NMI = 1			**Flags** O D I T S Z A P C 0 0
Operanden	**Benötigte Taktzyklen**	**Anzahl der Datenübertragungen***	**Bytes**	**Beispiel**
Keine Operanden	50	5	N/A	N/A

* Addieren Sie beim 8086 vier Taktzyklen für jede Übergabe eines 16-Bit-Wortes, das an einer ungeraden Adresse beginnt. Beim 8088 müssen Sie für jede Übertragung eines 16-Bit-Wortes vier Taktzyklen zusätzlich einkalkulieren.
†NMI ist kein Befehl; es dient hier nur zur Information über Zeitabläufe.

Mnemonics ©Intel, 1978

MOV	**MOV** Ziel, Quelle Byte oder Wort übertragen			**Flags** O D I T S Z A P C
Operanden	**Benötigte Taktzyklen**	**Anzahl der Datenübertragungen***	**Bytes**	**Beispiel**
Speicher, Akkumulator	10	1	3	MOV ARRAY [SI], AL
Akkumulator, Speicher	10	1	3	MOV AX, TEMP_RESULT
Register, Register	2	–	2	MOV AX, CX
Register, Speicher	8 + EA	1	2-4	MOV BP, STACK_TOP
Speicher, Register	9 – EA	1	2-4	MOV COUNT [DI], CX
Register, unmittelbar	4	–	2-3	MOV CL, 2
Speicher, unmittelbar	10 + EA	1	3-6	MOV MASK [BX] [SI], 2CH
Segmentregister, Register (16 Bit)	2	–	2	MOV ES, CX
Segmentregister, Speicher (16 Bit)	8 + EA	1	2-4	MOV DS, SEGMENT_BASE
Register (16 Bit), Segmentregister	2	–	2	MOV BP, SS
Speicher, Segmentregister	9 + EA	1	2-4	MOV [BX], SEG_SAVE, CS

MOVS	**MOVS** Ziel-String, Quell-String Zeichenkette übertragen			**Flags** O D I T S Z A P C
Operanden	**Benötigte Taktzyklen**	**Anzahl der Datenübertragungen***	**Bytes**	**Beispiel**
Ziel-String, Quell-String	18	2	1	MOVS LINE EDIT_DATA
(wiederhole) Ziel-Strng, Quell-Strng	9 + 17 Wdhlg	2 Wdhlg	1	REP MOVS SCREEN, BUFFER

MOVSB/MOVSW	**MOVSB/MOVSW** Keine Operanden Zeichenkette übertragen (Byte oder Wort)			**Flags** O D I T S Z A P C
Operanden	**Benötigte Taktzyklen**	**Anzahl der Datenübertragungen***	**Bytes**	**Beispiel**
Keine Operanden	18	2	1	MOVSB
(wiederhole) (keine Operanden)	9 – 17 Wdhlg	2 Wdhlg	1	REP MOVSW

MUL	**MUL** Quelle Multiplizieren			**Flags** O D I T S Z A P C X U U U U X
Operanden	**Benötigte Taktzyklen**	**Anzahl der Datenübertragungen***	**Bytes**	**Beispiel**
Register (8 Bit)	70-77	–	2	MUL BL
Register (16 Bit)	118-133	–	2	MUL CX
Speicher (8 Bit)	(76-83) + EA	1	2-4	MUL MONTH [SI]
Speicher (16 Bit)	(124-139) + EA	1	2-4	MUL BAUD_RATE

NEG	**NEG** Ziel Negieren			**Flags** O D I T S Z A P C X X X X X 1*
Operanden	**Benötigte Taktzyklen**	**Anzahl der Datenübertragungen***	**Bytes**	**Beispiel**
Register	3	–	2	NEG AL
Speicher	16 + EA	2	2-4	NEG MULTIPLIER

0 wenn Ziel = 0

* Addieren Sie beim 8086 vier Taktzyklen für jede Übergabe eines 16-Bit-Wortes, das an einer ungeraden Adresse beginnt. Beim 8088 müssen Sie für jede Übertragung eines 16-Bit-Wortes vier Taktzyklen zusätzlich einkalkulieren.

Mnemonics © Intel, 1978

NOP	NOP Keine Operation Keine Operation			Flags O D I T S Z A P C
Operanden	**Benötigte Taktzyklen**	**Anzahl der Datenübertragungen***	**Bytes**	**Beispiel**
Keine Operanden	3	–	1	NOP

NOT	NOT Ziel logisch invertieren			Flags O D I T S Z A P C
Operanden	**Benötigte Taktzyklen**	**Anzahl der Datenübertragungen***	**Bytes**	**Beispiel**
Register	3	–	2	NOT AX
Speicher	16 + EA	2	2-4	NOT CHARACTER

OR	OR Ziel, Quelle Logisches ODER bilden			Flags O D I T S Z A P C 0 X X U X 0
Operanden	**Benötigte Taktzyklen**	**Anzahl der Datenübertragungen***	**Bytes**	**Beispiel**
Register, Register	3	–	2	OR AL, BL
Register, Speicher	9 + EA	1	2-4	OR DX, PORT ID [DI]
Speicher, Register	16 + EA	2	2-4	OR FLAG BYTE, CL
Akkumulator, unmittelbar	4	–	2-3	OR AL,01101100B
Register, unmittelbar	4	–	3-4	OR CX,01H
Speicher, unmittelbar	17 + EA	2	3-6	OR [BX].CMD WORD 0CFH

OUT	OUT Anschluß, Akkumulator Byte oder Wort ausgeben			Flags O D I T S Z A P C
Operanden	**Benötigte Taktzyklen**	**Anzahl der Datenübertragungen***	**Bytes**	**Beispiel**
unmittelbar (8 Bit), Akkumulator	10	1	2	OUT 44, AX
DX, Akkumulator	8	1	1	OUT DX, AL

POP	POP Ziel Vom Stapel entnehmen			Flags O D I T S Z A P C
Operanden	**Benötigte Taktzyklen**	**Anzahl der Datenübertragungen***	**Bytes**	**Beispiel**
Register	8	1	1	POP DX
Segmentregister (außer CS)	8	1	1	POP DS
Speicher	17 + EA	2	2-4	POP PARAMETER

POPF	POPF Keine Operanden Kennzeichen vom Stapel laden			Flags O D I T S Z A P C R R R R R R R R R
Operanden	**Benötigte Taktzyklen**	**Anzahl der Datenübertragungen***	**Bytes**	**Beispiel**
Keine Operanden	8	1	1	POPF

* Addieren Sie beim 8086 vier Taktzyklen für jede Übergabe eines 16-Bit-Wortes, das an einer ungeraden Adresse, beginnt. Beim 8088 müssen Sie für jede Übertragung eines 16-Bit-Wortes vier Taktzyklen zusätzlich einkalkulieren.

Mnemonics ©Intel, 1978

PUSH	**PUSH** Quelle Ein Wort auf dem Stapel ablegen			**Flags** O D I T S Z A P C
Operanden	**Benötigte Taktzyklen**	**Anzahl der Datenübertragungen***	**Bytes**	**Beispiel**
Register	11	1	1	PUSH SI
Segmentregister (CS möglich)	10	1	1	PUSH ES
Speicher	16+EA	2	2-4	PUSH RETURN_CODE [SI]

PUSHF	**PUSHF** Keine Operanden Kennzeichenbits auf dem Stapel ablegen			**Flags** O D I T S Z A P C
Operanden	**Benötigte Taktzyklen**	**Anzahl der Datenübertragungen***	**Bytes**	**Beispiel**
Keine Operanden	10	1	1	PUSHF

RCL	**RCL** Ziel, Anzahl Durch Übertrag links rotieren			**Flags** O D I T S Z A P C X X
Operanden	**Benötigte Taktzyklen**	**Anzahl der Datenübertragungen***	**Bytes**	**Beispiel**
Register, 1	2	–	2	RCL CX, 1
Register, CL	8+4 Bit	–	2	RCL AL, CL
Speicher, 1	15+EA	2	2-4	RCL ALPHA, 1
Speicher, CL	20+EA+ 4 Bit	2	2-4	RCL [BP]. PARM CL

RCR	**RCR** Ziel, Anzahl Durch Übertrag rechts rotieren			**Flags** O D I T S Z A P C X X
Operanden	**Benötigte Taktzyklen**	**Anzahl der Datenübertragungen***	**Bytes**	**Beispiel**
Register, 1	2	–	2	RCR BX, 1
Register, CL	8+4 Bit	–	2	RCR BL, CL
Speicher, 1	15+EA	2	2-4	RCR [BX].STATUS, 1
Speicher, CL	20+EA+ 4 Bit	2	2-4	RCR ARRAY [DI]. CL

REP	**REP** Keine Operanden Wiederholen			**Flags** O D I T S Z A P C
Operanden	**Benötigte Taktzyklen**	**Anzahl der Datenübertragungen***	**Bytes**	**Beispiel**
Keine Operanden	2	–	1	REP MOVS DEST, SRCE

REPE/REPZ	**REPE/REPZ** Keine Operanden Wiederholen, solange gleich / solange Null			**Flags** O D I T S Z A P C
Operanden	**Benötigte Taktzyklen**	**Anzahl der Datenübertragungen***	**Bytes**	**Beispiel**
Keine Operanden	2	–	1	REPE CMPS DATA, KEY

* Addieren Sie beim 8086 vier Taktzyklen für jede Übergabe eines 16-Bit-Wortes, das an einer ungeraden Adresse beginnt. Beim 8088 müssen Sie für jede Übertragung eines 16-Bit-Wortes vier Taktzyklen zusätzlich einkalkulieren.

Mnemonics ©Intel, 1978

REPNE/REPNZ	REPNE/REPNZ Keine Operanden Wiederholen, solange n gleich, solange n. Null			Flags O D I T S Z A P C
Operanden	**Benötigte Taktzyklen**	**Anzahl der Datenübertragungen***	**Bytes**	**Beispiel**
Keine Operanden	2	–	1	REPNE SCAS INPUT LINE

RET	RET optionaler Stapelkorrekturwert Von Prozedur zurückkehren			Flags O D I T S Z A P C
Operanden	**Benötigte Taktzyklen**	**Anzahl der Datenübertragungen***	**Bytes**	**Beispiel**
intrasegment, kein Stapelkorrekturwert	8	1	1	RET
intrasegment, Stapelkorrekturwert	12	1	3	RET 4
intersegment, kein Stapelkorrekturwert	18	2	1	RET
intersegmnet, Stapelkorrekturwert	17	2	3	RET 2

ROL	ROL Ziel, Anzahl Links rotieren			Flags O D I T S Z A P C X X
Operanden	**Benötigte Taktzyklen**	**Anzahl der Datenübertragungen***	**Bytes**	**Beispiel**
Register, 1	2	–	2	ROL BX, 1
Register, CL	8+4 Bit	–	2	ROL DI,CL
Speicher, 1	15+EA	2	2-4	ROL FLAG_BYTE [DI],1
Speicher, CL	20+EA+ 4 Bit	2	2-4	ROL ALPHA, CL

ROR	ROR Ziel, Anzahl Rechts rotieren			Flags O D I T S Z A P C X X
Operanden	**Benötigte Taktzyklen**	**Anzahl der Datenübertragungen***	**Bytes**	**Beispiel**
Register, 1	2	–	2	ROR AL, 1
Register, CL	8+4/Bit	–	2	ROR BX, CL
Speicher, 1	15+EA	2	2-4	ROR PORT_STATUS, 1
Speicher, CL	20+EA+ 4/Bit	2	2-4	ROR CMD_WORD, CL

SAHF	SAHF Keine Operanden Register AH ins Kennzeichenregister übertragen			Flags O D I T S Z A P C R R R R R
Operanden	**Benötigte Taktzyklen**	**Anzahl der Datenübertragungen***	**Bytes**	**Beispiel**
Keine Operanden	4	–	1	SAHF

SAL/SHL	SAL/SHL Ziel, Anzahl Arithmetisch links schieben / logisch links schieben			Flags O D I T S Z A P C X X
Operanden	**Benötigte Taktzyklen**	**Anzahl der Datenübertragungen***	**Bytes**	**Beispiel**
Register, 1	2	–	2	SAL AL,1
Register, CL	8+4/Bit	–	2	SHL DI,CL
Speicher, 1	15+EA	2	2-4	SHL [BX].OVERDRAW, 1
Speicher, CL	20+EA+ 4/Bit	2	2-4	SAL STORE COUNT,CL

* Addieren Sie beim 8086 vier Taktzyklen für jede Übergabe eines 16-Bit-Wortes, das an einer ungeraden Adresse beginnt. Beim 8088 müssen Sie für jede Übertragung eines 16-Bit-Wortes vier Taktzyklen zusätzlich einkalkulieren.
Mnemonics ©Intel, 1978

SAR	SAR Ziel, Quelle Arithmetisch rechts schieben			Flags	O D I T S Z A P C X X X U X X
Operanden	**Benötigte Taktzyklen**	**Anzahl der Datenübertragungen***	**Bytes**	**Beispiel**	
Register, 1	2	–	2	SAR DX, 1	
Register, CL	8 + 4 Bit	–	2	SAR DI, CL	
Speicher, 1	15 + EA	2	2-4	SAR N BLOCKS 1	
Speicher, CL	20 + EA + 4 Bit	2	2-4	SAR N BLOCKS CL	

SBB	SBB Ziel, Quelle Mit Borger subtrahieren			Flags	O D I T S Z A P C X X X X X X
Operanden	**Benötigte Taktzyklen**	**Anzahl der Datenübertragungen***	**Bytes**	**Beispiel**	
Register, Register	3	–	2	SBB BX, CX	
Register, Speicher	9 + EA	1	2-4	SBB DI, [BX] PAYMENT	
Speicher, Register	16 + EA	2	2-4	SBB BALANCE, AX	
Akkumulator, unmittelbar	4	–	2-3	SBB AX, 2	
Register, unmittelbar	4	–	3-4	SBB CL 1	
Speicher, unmittelbar	17 + EA	2	3-6	SBB COUNT [SI], 10	

SCAS	SCAS Ziel-Strng Zeichenkette untersuchen			Flags	O D I T S Z A P C X X X X X X
Operanden	**Benötigte Taktzyklen**	**Anzahl der Datenübertragungen***	**Bytes**	**Beispiel**	
Ziel-Strng	15	1	1	SCAS INPUT LINE	
(wiederhole) Ziel-String	9 + 15 Wdhlg	1 Wdhlg	1	REPNE SCAS BUFFER	

SEGMENT†	SEGMENT Vorsatz Bis zum angebenenen Segment fortschreiten			Flags	O D I T S Z A P C
Operanden	**Benötigte Taktzyklen**	**Anzahl der Datenübertragungen***	**Bytes**	**Beispiel**	
Keine Operanden	2	–	1	MOV SS, PARAMETER, AX	

SHR	SHR Ziel, Anzahl Logisch rechts schieben			Flags	O D I T S Z A P C X X
Operanden	**Benötigte Taktzyklen**	**Anzahl der Datenübertragungen***	**Bytes**	**Beispiel**	
Register, 1	2	–	2	SHR SI, 1	
Register, CL	8 + 4 Bit	–	2	SHR SI, CL	
Speicher, 1	15 + EA	2	2-4	SHR ID BYTE [SI] [BX], 1	
Speicher, CL	20 + EA + 4 Bit	2	2-4	SHR INPUT WORD, CL	

SINGLE STEP†	SINGLE STEP Unterbrechung n. jedem Programmschritt, wenn Trap-Flag gesetzt			Flags	O D I T S Z A P C 0 0
Operanden	**Benötigte Taktzyklen**	**Anzahl der Datenübertragungen***	**Bytes**	**Beispiel**	
Keine Operanden	50	5	N/A	N/A	

* Addieren Sie beim 8086 vier Taktzyklen für jede Übergabe eines 16-Bit-Wortes, das an einer ungeraden Adresse beginnt. Beim 8088 müssen Sie für jede Übertragung eines 16-Bit-Wortes vier Taktzyklen zusätzlich einkalkulieren.
† SEGMENT ist kein eigenständiger Befehl. Es wird hier zur Information über Zeitabläufe angeführt.
† SINGLE STEP ist kein Befehl. Es dient hier nur zur Information über Zeitabläufe.
Mnemonics ©Intel, 1978

STC	STC Keine Operanden Übertragskennzeichen setzen			Flags O D I T S Z A P C 1
Operanden	**Benötigte Taktzyklen**	**Anzahl der Datenübertragungen***	**Bytes**	**Beispiel**
Keine Operanden	2	--	1	STC

STD	STD Keine Operanden Richtungskennzeichen setzen			Flags O D I T S Z A P C 1
Operanden	**Benötigte Taktzyklen**	**Anzahl der Datenübertragungen***	**Bytes**	**Beispiel**
Keine Operanden	2	--	1	STD

STI	STI Keine Operanden Unterbrechungsfreigabekennzeichen setzen			Flags O D I T S Z A P C 1
Operanden	**Benötigte Taktzyklen**	**Anzahl der Datenübertragungen***	**Bytes**	**Beispiel**
Keine Operanden	2	--	1	STI

STOS	STOS Ziel-String (Byte- oder Wort-) Zeichenkette speichern			Flags O D I T S Z A P C
Operanden	**Benötigte Taktzyklen**	**Anzahl der Datenübertragungen***	**Bytes**	**Beispiel**
Ziel-String	11	1	1	STOS PRINT LINE
(wiederhole) Ziel-String	9+10 Wdhlg	1 Wdhlg	1	REP STOS DISPLAY

SUB	SUB Ziel, Quelle Subtrahieren			Flags O D I T S Z A P C X X X X X
Operanden	**Benötigte Taktzyklen**	**Anzahl der Datenübertragungen***	**Bytes**	**Beispiel**
Register, Register	3	--	2	SUB CX, BX
Register, Speicher	9+EA	1	2-4	SUB DX, MATH_TOTAL [SI]
Speicher, Register	16+EA	2	2-4	SUB [BP+2], CL
Akkumulator, unmittelbar	4	--	2-3	SUB AL, 10
Register, unmittelbar	4	--	3-4	SUB SI, 5280
Speicher, unmittelbar	17+EA	2	3-6	SUB [BP]. BALANCE, 1000

TEST	TEST Ziel, Quelle Testen (logisch vergleichen)			Flags O D I T S Z A P C 0 X X U X 0
Operanden	**Benötigte Taktzyklen**	**Anzahl der Datenübertragungen***	**Bytes**	**Beispiel**
Register, Register	2	--	2	TEST SI, DI
Register, Speicher	9+EA	1	2-4	TEST SI,END_COUNT
Akkumulator, unmittelbar	4	--	2-3	TEST AL,00100000B
Register, unmittelbar	5	--	3-4	TEST BX, 0CC4H
Speicher, unmittelbar	11+EA	--	3-6	TEST RETURN_CODE, 01H

* Addieren Sie beim 8086 vier Taktzyklen für jede Übergabe eines 16-Bit-Wortes, das an einer ungeraden Adresse beginnt. Beim 8088 müssen Sie für jede Übertragung eines 16-Bit-Wortes vier Taktzyklen zusätzlich einkalkulieren.

Mnemonics ©Intel, 1978

WAIT	**WAIT** Keine Operanden Warten, bis Test-Eingang der CPM aktiv wird			**Flags** O D I T S Z A P C
Operanden	**Benötigte Taktzyklen**	**Anzahl der Datenüber-tragungen***	**Bytes**	**Beispiel**
Keine Operanden	3+5n	–	1	WAIT

XCHG	**XCHG** Ziel, Quelle Austauschen			**Flags** O D I T S Z A P C
Operanden	**Benötigte Taktzyklen**	**Anzahl der Datenüber-tragungen***	**Bytes**	**Beispiel**
Akkumulator, Register (16 Bit)	3	–	1	XCHG AX, BX
Speicher, Register	17+EA	2	2-4	XCHG SEMAPHORE, AX
Register, Register	4	–	2	XCHG AL, BL

XLAT	**XLAT** Quell-Tabelle Übersetzen			**Flags** O D I T S Z A P C
Operanden	**Benötigte Taktzyklen**	**Anzahl der Datenüber-tragungen***	**Bytes**	**Beispiel**
Quell-Tabelle	11	1	1	XLAT ASCII_TAB

XOR	**XOR** Ziel, Quelle Exclusives ODER bilden			**Flags** O D I T S Z A P C 0 X X U X 0
Operanden	**Benötigte Taktzyklen**	**Anzahl der Datenüber-tragungen***	**Bytes**	**Beispiel**
Register, Register	3	–	2	XOR CX, BX
Register, Speicher	9 + EA	1	2-4	XOR CL, MASK_BYTE
Speicher, Register	16 + EA	2	2-4	XOR ALPHA[SI], DX
Akkumulator, unmittelbar	4	–	2-3	XOR AL, 01000010B
Register, unmittelbar	4	–	3-4	XOR SI, 00C2H
Speicher, unmittelbar	17+EA	2	3-6	XOR RETURN_CODE, 0D2H

* Addieren Sie beim 8086 vier Taktzyklen für jede Übergabe eines 16-Bit-Wortes, das an einer ungeraden Adresse beginnt. Beim 8088 müssen Sie für jede Übertragung eines 16-Bit-Wortes vier Taktzyklen zusätzlich einkalkulieren.

Mnemonics ©Intel, 1978

Stichwortverzeichnis

Arithmetische Programme, 175–188
Assembler, 270–280
Assemblerdirektiven, 276
Assemblerfelder, 270, 271
Assemblersprache, 273–278
  Konstante oder Literale, 276
  Marken, 274, 275
  Operatoren, 276
  Programmieren in -, 262, 263
  Symbole, 274
Ausführungseinheit, 47–49
Ausgabeanschluß, 8086/8088, 217
Ausgabebefehl, Beispiel, 216, 217
Ausgerichtete und
  nichtausgerichtete Worte 59, 60

BCD-Addition, 8-Bit, 181–184
BCD-Arithmetik, 181–185
BCD-Darstellung, 28–30
BCD-Subtraktion, 184, 185
BHE, 55
BIOS
  Definition, 239
  Parameterübergabe, 241–243
  Präambel, 243
  Zugriff auf Peripheriegeräte
  mit dem -, 241–243
BIU, 47–49
BP-Register, 50
BX, BL, BH-Register, 50
Basiszeiger, 50
Befehle, Übersicht, 287
Befehlszeiger, 51
Befehlszyklus,
  allgemeine Definition, 44–47
Befehlszählerregister, 43
Binary Coded Decimal
  (binär codierte Dezimalziffer),
  Beispiel, 28, 29
Binäre Division, 187
Binäre Zahlen, 16–19, 33
  mit Vorzeichen, 18, 19
Bit, D-, 68, 69
Bit, MOD-, 68, 69
Bit, W-, 67, 68
Blockdiagramm der Register, 50, 51
Blockdiagramm des 8086/8088, 48
Busschnittstelleneinheit, 48, 49

CALL (Prozedur aufrufen), 83, 84
CALL, Intersegment-, 192
CALL, Intrasegment-, 191, 192
CALL-RETURN-
  Unstimmigkeit, 192
CBW (Byte zu Wort umwandeln), 85
CF (Übertragskennzeichen), 52
CLC (Übertragskennzeichen
  löschen), 86
CLD (Richtungkennzeichen
  löschen), 87
CLI (Unterbrechungsfreigabe-
  kennzeichen löschen), 88
CMC (Übertragungskennzeichen
  komplementieren), 89
CMP (Vergleichen), 90
CMPS (Zeichenkette
  vergleichen), 91
CPU zurücksetzen, 207, 208
CS (Codesegment), 51
CWD (Wort zu Doppelwort
  umwandeln), 92
CX, CH, CL-Register, 50
Code, Objekt-, 66–71

D-Bit, 68, 69
DAA (Dezimale Anpassung für
  Addition), 93
DAS (Dezimale Anpassung für
  Subtraktion), 94
DEC (Dekrementieren), 95
DF (Richtung), 52
DIV (Dividieren), 96, 97
DX, DH, DL-Register, 50
Darstellung von Information, 14–36

Daten
alphanumerische, 31–33
binäre, 16–19, 33
binäre mit Vorzeichen, 18, 19
hexadezimale, 33–36
numerische, 14–16
oktale, 33–36
Dezimal-zu-BCD-Umwandlungstabelle, 285
Dezimal-zu-Binär-Umwandlung, 16
Direkte Adressierung, 63
Direktiven, Assembler-, 276, 278
Division
binäre, 187
mit dem 8086/8088, 187
Drucker-E/A, 243, 244

E/A, 209–238
E/A, Definition, 209, 210
E/A-Adressierung, 211–213
E/A-Beispiel, 223–233
E/A-Einrichtung, 213, 214
E/A-Raum, reservierter, 213
ESC (Objektcode und Operand für Koprozessor bereitstellen), 98
EU (Ausführungseinheit), 48, 49
EX (Extrasegment), 51
Ein- und Ausgabe, 209–238
Ein- und Ausgabe, Definition, 209, 210
Ein- und Ausgabe-Adressierung, 211–213
Ein- und Ausgabe-Raum, reservierter, 213
Einer-Komplement, 19, 20
Eingabeanschluß, 8086/8088, 217
Eingabebefehl, Beispiel, 214
Einplatinen-Mikrocomputer, 268
Entwicklungssystem, 268
Erzeugen eines Signals mit Hilfe von Software, 219
Erzeugung der Befehlsadressen, 61, 62
Erzeugung eines Impulses mit Hilfe von Software, 220, 221
Erzeugung von Adressen, 61

Fehlermeldungen, Assembler, 273
Feld, R/M-, 68
Feld, REG-, 67
Flußdiagramm, Definition, 12, 13

Gleitpunktdarstellung, 30, 31
Grund-E/A-System, Definition, 242

HLT (Halt), 99
Hardwarealternativen, 268–269
Hardwarebetriebsmittel, Zusammenfassung, 269
Hexadezimale Umwandlungstabelle, 281
Hexadezimale Daten, 33–36
Holphase, Definition, 45

IDIV (Ganzzahlig dividieren), 100
IF (Unterbrechungsfreigabekennzeichen), 52
IMUL (Ganzzahlig multiplizieren), 102
IN (Einlesen), 102
INC (Inkrementieren), 104
INT (Unterbrechen), 105
INTO (Bei gesetztem Überlaufskenn zeichen unterbrechen), 106
INTR, 202
IRET (Rückkehr von Unterbrechung), 107
Indexregister, Definition, 43
Information, Darstellung von, 14–36
Interne Organisation des 8086/8088, 47–54
Interne Unterbrechungen, 204, 205
Interner Aufbau des Mikroprozessors, 40–44
Intersegment-Aufruf, 192
Intrasegment-Aufruf, 191, 192

JA (Sprung bei vorzeichenlos größer), 108
JAE (Sprung bei vorzeichenlos größer oder gleich), 109
JB (Sprung bei vorzeichenlos kleiner), 110
JBE (Sprung bei vorzeichenlos kleiner oder gleich), 111
JC (Sprung bei gesetztem Übertragskennzeichen), 112

JCXZ (Sprung, falls CX-Register null), 113
JE (Sprung bei gleich), 114
JG (Sprung bei größer), 115
JGE (Sprung bei größer oder gleich), 116
JL (Sprung bei kleiner), 117
JLE (Sprung bei kleiner oder gleich), 118
JMP (Unbedingter Sprung), 119, 120
JNA (Sprung bei vorzeichenlos nicht größer), 111
JNAE (Sprung bei vorzeichenlos nicht größer oder gleich), 110
JNB (Sprung bei vorzeichenlos nicht kleiner), 109
JNBE (Sprung bei vorzeichenlos nicht kleiner oder gleich), 108
JNC (Sprung bei nicht gesetzem Übertragskennzeichen), 121
JNE (Sprung bei nicht gleich), 122
JNG (Sprung bei nicht größer), 118
JNGE (Sprung bei nicht größer oder gleich), 117
JNL (Sprung bei nicht kleiner), 116
JNLE (Sprung bei nicht kleiner oder gleich), 115
JNO (Sprung bei nicht gesetztem Überlaufskennzeichen), 123
JNP (Sprung bei nicht gesetztem Paritätskennzeichen), 125
JNS (Sprung bei nicht gesetztem Vorzeichenkennzeichen), 124
JNZ (Sprung bei nicht null), 122
JO (Sprung bei gesetztem Überlaufskennzeichen), 126
JP (Sprung bei gesetztem Paritätskennzeichen), 127
JPE (Sprung bei gerader Parität), 127
JPO (Sprung bei ungerader Parität), 125
JS (Sprung bei gesetztem Vorzeichenkennzeichen), 128
JZ (Sprung bei null), 114

Kennzeichenbits für den 8086/8088, 51–53
Kennzeichenbits und Register, Blockdiagramm, 52, 53

LAHF (Register AH mit Kennzeichenbits laden), 129
LDS (Zeiger einschließlich DS laden), 130
LEA (Effektive Adresse laden), 131
LES (Zeiger einschließlich ES laden), 132
LOCK (LOCK-Signal auf dem Bus durchsetzen), 133
LODS (Zeichenkette laden), 134
LOOP (Schleife, falls CX nicht Null), 135
LOOPE (Schleife, solange gleich), 136
LOOPNE (Schleife, solange nicht gleich), 137
LOOPNZ (Schleife, solange nicht Null), 137
LOOPZ (Schleife, solange Null), 136
Lesen von 16 Bits an gerader Adresse, 57
Lesen von 16 Bits an ungerader Adresse, 58
Lesen von 8 Bits an gerader Adresse, 58
Lesen von 8 Bits an ungerader Adresse, 58

MOD-Bit, 68, 69
MOV (Byte oder Wort übertragen), 138
MOVS (Zeichenkette übertragen), 139
MUL (Multiplizieren), 144
Mikrocomputer, Einplatinen-, 269
Mikrocomputer, Hobby-, 269
Mikroprozessor, Systemarchitektur, 37–40
Multiplikation, 185, 186

NEG (Negieren – Zweier–komplement bilden), 141
NMI, 196–202
NOP (Keine Operation), 142
NOT (Logisch invertieren), 143
Nichtausgerichtete Worte, 59, 60
Nichtmaskierbare Unterbrechungen, 196, 202

Numerische Daten,
Darstellung, 14–16

OF (Überlaufskennzeichen), 52
OR (Logisches Oder bilden), 144
OUT (Ausgeben), 145
Objektcode, 66–71
für Index- und Basisregister, 70
Oktale Daten, 33–36
Operatoren, Assemblersprache, 276

PF (Paritätskennzeichen), 52
PIO, 8255, 223–233
PIO, Überblick, 225, 226
POP, (Vom Stapel entnehmen), 146
POPF, (Kennzeichenbits vom
Stapel laden), 147
PUSH, (Daten auf dem Stapel
ablegen), 148
Parameterübergabe BIOS, 241–243
Programm, 8088-, 258
Programme, arithmetische, 175–188
Programmieren in einer höheren
Programmiersprache, 263
Programmierung eines
PIO-Bausteins, 228–233
Programmierung, Definition, 11
Programmierungsmöglichkeiten,
allgemeine, 262–263
Präambel, BIOS, 243, 244

Quellindexregister, 50

R/M-Feld, 68, 69
RCL (Durch Übertrag links
rotieren), 150
RCR (Durch Übertrag rechts
rotieren), 151
REG-Feld, 67
REP (Wiederholen), 152
REPE (Wiederholen,
solange gleich), 152
REPNE (Wiederholen,
solange nicht gleich), 153
REPNZ (Wiederholen,
solange nicht Null), 153
REPZ (Wiederholen,
solange Null), 152
RET (Von Prozedur
zurückkehren), 154, 155
ROL (Links rotieren), 156
ROR (Rechts rotieren), 157
RS-232, E/A, 243–255
Raum, für E/A reserviert, 213
Register und Kennzeichen,
Blockdiagramm, 53
Register, interne des 8086/8088,
49–54
Registeradressierung, 63–66
Registerindirekte Adressierung mit
Basis + Index + Konstante, 65, 66
Registerindirekte Adressierung mit
Basis und Index, 65
Registerindirekte Adressierung mit
Verschiebeanteil, 65
Registerindirekte Adressierung,
64, 65
Rücksprung von Unterbrechung,
205, 206

SAHF (Register AH ins Kenn–
zeichenregister übertragen), 158
SAL (Arithmetisch
links schieben), 159
SAR (Arithmetisch
rechts schieben), 160
SBB (Mit geborgtem Übertrag
subtrahieren), 162
SCAS (Byte- oder Wort-Zeichen-
kette untersuchen), 163
SF (Vorzeichenkennzeichen), 52
SHL (Logisch links schieben), 159
SHR (Logisch rechts schieben), 164
SP-Register (Stack Pointer,
Stapelzeiger), 50
SS (Stapelsegment), 51
STC (Übertagskennzeichen
setzen), 165
STD (Richtungskennzeichen
setzen), 166
STI (Unterbrechungsfreigabe-
kennzeichen setzen), 167
STOS (Byte- oder Wort-
Zeichenkette speichern), 168
SUB (Subtrahieren), 169
Sechzehn-Bit-Addition, 179
Sechzehn-Bit-Multiplikation, 186
Sechzehn-Bit-Subtraktion, 179–181
Software-Assembler, 270–280
Software-Unterbrechung, 204, 205

Software-Unterstützung, 264, 266
Softwareentwicklung,
  Stufen der, 267
Speicherorganisation des
  8086/8088, 55–63
Stapel, Definition, 43
Stapelzeiger, Definition, 43
Stapelzeigerregister, 50
Stufen der Softwareentwicklung, 267
Subtraktion, 16-Bit, 179–181
Subtraktion, BCD 8-Bit, 184, 185
Symbole, Assemblersprache, 274

TEST (Logisch vergleichen), 170
TF (Trapkennzeichen), 52
Tabelle, Unterbrechungs, 199, 200
Tastatur, E/A, 250–253
Tastatur-Abtastprogramm, 231–233

Überlauf, Beispiele, 24–27
Übersicht über 8086/8088
  Befehle, 287
Übertrag, Beispiele, 23–27
Umwandlungstabelle
  ASCII, 283
  Dezimal-zu-BCD, 285
  Hexadezimale, 281
Unmittelbare Adressierung, 63
Unstimmigkeit,
  CALL-RETURN-, 192
Unterbrechung, Definition, 195
Unterbrechung, INTR, 202
Unterbrechung, Rückkehr von, 107
Unterbrechung, Software-, 204–207
Unterbrechung,
  allgemeine Information, 195–197
Unterbrechung, interne, 204, 205
Unterbrechung,
  nichtmaskierbare, 186, 187
Unterbrechung, reservierte, 201, 202
Unterbrechungstabelle, 199
Unterprogramme, 188–194
Unterstützung, Software-, 264–266

Variable Adresse, Adressierung
  eines Anschlusses mit, 214, 215
Verbindung PIO – CPU, 227, 228
Verzögerung, Software zur
  Erzeugung einer, 220

W-Bit, 67, 68
WAIT (Warten), 171
Wort, ausgerichtet, 59, 60

XCHG (Austauschen), 172
XLAT (Übersetzen), 173
XOR (Exklusives Oder bilden), 174

ZF (Nullkennzeichen), 52
Zeitgeberbaustein, 8253, 234–237
Zielindex, 50
Zurücksetzen (RESET) des
  8086/8088, 207, 208
Zweier-Komplement, 20–33
Zyklus,
  Definition des Befehls-, 44–47

Die SYBEX Bibliothek

EINFÜHRUNG IN PASCAL UND UCSD/PASCAL
von Rodnay Zaks – das Buch für jeden, der die Programmiersprache PASCAL lernen möchte. Vorkenntnisse in Computerprogrammierung werden nicht vorausgesetzt. Eine schrittweise Einführung mit vielen Übungen und Beispielen. 535 Seiten, 130 Abbildungen, Ref.Nr.: **3004** (1982)

DAS PASCAL HANDBUCH
von Jacques Tiberghien – ein Wörterbuch mit jeder Pascal-Anweisung und jedem Symbol, reservierten Wort, Bezeichner und Operator, für beinahe alle bekannten Pascal-Versionen. 480 Seiten, 270 Abbildungen, Format 23 x 18 cm, Ref.Nr.: **3005** (1982)

PROGRAMMIERUNG DES Z80
von Rodnay Zaks – ein kompletter Lehrgang in der Programmierung des Z80 Mikroprozessors und eine gründliche Einführung in die Maschinensprache. 608 Seiten, 176 Abbildungen, Format DIN A5, Ref.Nr.: **3006** (1982)

PASCAL PROGRAMME – MATHEMATIK, STATISTIK, INFORMATIK
von Alan Miller – eine Sammlung von 60 der wichtigsten wissenschaftlichen Algorithmen samt Programmauflistung und Musterdurchlauf. Ein wichtiges Hilfsmittel für Pascal-Benutzer mit technischen Anwendungen. 398 Seiten, 120 Abbildungen, Format 23 x 18 cm, Ref.Nr.: **3007** (1982)

POCKET MIKROCOMPUTER LEXIKON
– die schnelle Informations-Börse! 1300 Definitionen, Kurzformeln, technische Daten, Lieferanten-Adressen, ein englisch-deutsches und französisch-deutsches Wörterbuch. 176 Seiten, Format DIN A6, Ref.Nr. **3008** (1982)

PROGRAMMIERUNG DES 6502 (2. überarbeitete Ausgabe)
von Rodnay Zaks – Programmierung in Maschinensprache mit dem Mikroprozessor 6502, von den Grundkonzepten bis hin zu fortgeschrittenen Informationsstrukturen. 368 Seiten, 160 Abbildungen, Format DIN A5, Ref.Nr.: **3011** (1982)

MIKROPROZESSOR INTERFACE TECHNIKEN (3. überarbeitete Ausgabe)
von Rodnay Zaks/Austin Lesea – Hardware- und Software-Verbindungstechniken samt Digital/Analog-Wandler, Peripheriegeräte, Standard-Busse und Fehlersuchtechniken. 432 Seiten, 400 Abbildungen, Format DIN A5, Ref.Nr.: **3012** (1982)

VORSICHT! Computer brauchen Pflege
von Rodnay Zaks – das Buch, das Ihnen die Handhabung eines Computersystems erklärt – vor allem, was Sie damit nicht machen sollten. Allgemeingültige Regeln für die pflegliche Behandlung Ihres Systems. 240 Seiten, 96 Abbildungen, Ref.Nr.: **3013** (1983)

6502 ANWENDUNGEN
von Rodnay Zaks – das Eingabe-/Ausgabe-Buch für Ihren 6502-Microprozessor. Stellt die meistgenutzten Programme und die dafür notwendigen Hardware-Komponenten vor. 288 Seiten, 213 Abbildungen, Ref.Nr.: **3014** (1983)

BASIC PROGRAMME – MATHEMATIK, STATISTIK, INFORMATIK
von Alan Miller – eine Bibliothek von Programmen zu den wichtigsten Problemlösungen mit numerischen Verfahren, alle in BASIC geschrieben, mit Musterlauf und Programmlisting. 352 Seiten, 147 Abbildungen, Ref.Nr.: **3015** (1983)

BASIC ÜBUNGEN FÜR DEN APPLE
von J.-P. Lamoitier – das Buch für APPLE-Nutzer, die einen schnellen Zugang zur Programmierung in BASIC suchen. Abgestufte Übungen mit zunehmendem Schwierigkeitsgrad. 256 Seiten, 190 Abbildungen, Ref.Nr.: **3016** (1983)

CHIP UND SYSTEM: Einführung in die Mikroprozessoren-Technik
von Rodnay Zaks – eine sehr gut lesbare Einführung in die faszinierende Welt der Computer, vom Microprozessor bis hin zum vollständigen System. 576 Seiten, 325 Abbildungen, Ref.Nr.: **3017** (1984)

EINFÜHRUNG IN WORDSTAR
von Arthur Naiman – eine klar gegliederte Einführung, die aufzeigt, wie das Textbearbeitungsprogramm WORDSTAR funktioniert, was man damit tun kann und wie es eingesetzt wird. 240 Seiten, 36 Abbildungen, Ref.Nr.: **3019** (1983)

MEIN SINCLAIR ZX81
von D. Hergert – eine gut lesbare Einführung in diesen Einplatinencomputer und dessen Programmierung in BASIC. 176 Seiten, 47 Abbildungen, Ref: **3021** (1983)

SINCLAIR ZX SPECTRUM Programme zum Lernen und Spielen
von T. Hartnell – ein Buch zur praktischen Anwendung. Grundzüge des Programmierens aus dem kaufmännischen Bereich sowie Spiele, Lehr- und Lernprogramme in BASIC. 232 Seiten, 140 Abbildungen, Ref. Nr. **3022** (1983)

BASIC ÜBUNGEN FÜR DEN IBM PERSONAL COMPUTER
von J.-P. Lamoitier – vermittelt Ihnen BASIC durch praktische und umfassende Übungen anhand von realistischen Programmen: Datenverarbeitung, Statistik, kommerzielle Programme, Spiele u.v.m. 256 Seiten, 192 Abbildungen, Ref.-Nr.: **3023** (1983)

PROGRAMMSAMMLUNG ZUM IBM PERSONAL COMPUTER
von S. R. Trost – mehr als 65 getestete, direkt einzugebende Anwenderprogramme, die eine weite Palette von kaufmännischen, persönlichen und schulischen Anwendungen abdecken. 192 Seiten, 158 Abbildungen, Ref.-Nr.: **3024** (1983)

PLANEN UND ENTSCHEIDEN MIT BASIC
von X. T. Bui – eine Sammlung von interaktiven, kommerziell-orientierten BASIC-Programmen für Management- und Planungsentscheidungen. 200 Seiten, 53 Abbildungen, Ref.-Nr.: **3025** (1983)

BASIC FÜR DEN KAUFMANN
von D. Hergert – das BASIC-Buch für Studenten und Praktiker im kaufmännischen Bereich. Enthält Anwendungsbeispiele für Verkaufs- und Finanzberichte, Grafiken, Abschreibungen u.v.m. 208 Seiten, 85 Abbildungen, Ref.-Nr.: **3026** (1983)

SINCLAIR ZX SPECTRUM BASIC HANDBUCH
von D. Hergert – eine wichtige Hilfe für jeden SPECTRUM-Anwender. Gibt eine Übersicht aller BASIC-Begriffe, die auf diesem Rechner verwendet werden können, und erläutert sie ausführlich anhand von Beispielen. 288 Seiten, 188 Abbildungen, Ref.-Nr.: **3027** (1983)

SINCLAIR ZX81 BASIC HANDBUCH
von D. Hergert – vermittelt Ihnen das vollständige BASIC-Vokabular anhand von praktischen Beispielen, macht Sie zum Programmierer Ihres ZX81. 181 Seiten, 120 Abbildungen, Ref.-Nr.: **3028** (1983)

PROGRAMME FÜR MEINEN APPLE II
von S. R. Trost – enthält eine Reihe von lauffähigen Programmen samt Listing und Beispiellauf. Hilft Ihnen, viele neue Anwendungen für Ihren APPLE II zu entdecken und erfolgreich einzusetzen. 192 Seiten, 158 Abbildungen, Ref.-Nr.: **3029** (1983)

ERFOLG MIT VisiCalc
von D. Hergert – umfassende Einführung in VisiCalc und seine Anwendung. Zeigt Ihnen u. a.: Aufstellung eines Verteilungsbogens, Benutzung von VisiCalc-Formeln, Verwendung der DIF-Datei-Funktion. 224 Seiten, 58 Abbildungen, Ref.-Nr.: **3030** (1983)

APPLE II LEICHT GEMACHT
von J. Kascmer – macht Sie schnell mit Tastatur, Bildschirm und Diskettenlaufwerken vertraut. Sie lernen, wie leicht es ist, Ihr eigenes BASIC-Programm zu schreiben. Ca. 176 Seiten, mit Abbildungen, Ref.-Nr.: **3031** (April 1984)

PLANEN, KALKULIEREN, KONTROLLIEREN MIT BASIC-TASCHENRECHNERN
von P. Ickenroth – präsentiert eine Reihe von direkt anwendbaren BASIC-Programmen für zahlreiche kaufmännische Berechnungen mit Ihrem BASIC-Taschenrechner. 144 Seiten, 48 Abbildungen, Ref.-Nr.: **3032** (1983)

MEIN ERSTES BASIC PROGRAMM
von Rodnay Zaks – das Buch für Einsteiger! Viele farbige Illustrationen und leichtverständliche Diagramme bringen Spaß am Lernen. In wenigen Stunden schreiben Sie Ihr erstes nützliches Programm. 208 Seiten, illustriert, Ref.-Nr.: **3033** (1983)

IBM PC-DOS HANDBUCH
von R. A. King – umfassende Einführung in das Disketten-Betriebssystem Ihres IBM PC, seine grundsätzlichen Möglichkeiten und Funktionen sowie auch fortgeschrittene Funktionen (einschließlich der Version 2.0). Ca. 320 Seiten, ca. 50 Abbildungen, Ref.-Nr.: **3034** (Mai 1984)

APPLE II BASIC HANDBUCH
von D. Hergert – ein handliches Nachschlagewerk, das neben Ihren Apple II, II+ oder IIe stehen sollte. Dank vieler Tips und Vorschläge eine wesentliche Erleichterung fürs Programmieren. Ca. 272 Seiten, 116 Abbildungen, Ref.-Nr. **3036** (April 1984)

Z80 ANWENDUNGEN
von J. W. Coffron – vermittelt alle nötigen Anweisungen, um Peripherie-Bausteine mit dem Z80 zu steuern und individuelle Hardware-Lösungen zu realisieren. Ca. 320 Seiten, ca. 200 Abbildungen, Ref.-Nr.: **3037** (Juni 1984)

COMMODORE 64 – LEICHT GEMACHT
von J. Kascmer – führt Sie schnell in die Bedienung von Tastatur, Bildschirm und Diskettenlaufwerken ein, macht Sie zum BASIC-Programmierer Ihres C64! Ca. 176 Seiten, mit Abbildungen, Ref.-Nr.: **3038** (April 1984)

SPIELEN, LERNEN, ARBEITEN mit dem TI99/4A
von K.-J. Schmidt und G.-P. Raabe – eine eingehende Einführung in die Bedienung und Programmierung des TI99/4A. Mit den vielen Beispielprogrammen holen Sie das Beste aus Ihrem Computer heraus. 192 Seiten, 41 Abbildungen, Ref.-Nr.: **3039** (1984)

MEIN DRAGON 32
von N. Hesselmann – entwickelt Ihre Fähigkeiten in der Nutzung, Programmierung und erweiterten Anwendung Ihres Rechners anhand von vielen Beispielprogrammen. 256 Seiten, 41 Abbildungen, Ref.-Nr.: **3041** (1984)

ERFOLG MIT MULTIPLAN
von Th. Ritter – das Tabellenkalkulations-Programm Multiplan hilft Ihnen bei der Lösung kommerzieller, wissenschaftlicher und allgemeiner Probleme. Lernen Sie die Möglichkeiten kennen, Ihre Software optimal zu nutzen! Ca. 280 Seiten, ca. 60 Abbildungen, Ref.-Nr.: **3043** (Juni 1984)

FARBSPIELE MIT DEM COMMODORE 64
von W. Black und M. Richter – 20 herrliche Farbspiele für Ihren C64, mit Beschreibung, Programmlisten und Bildschirm-Darstellungen. Für mehr Freizeit-Spaß mit Ihrem Commodore! Ca. 200 Seiten, ca. 60 Abbildungen, Ref.-Nr.: **3044** (Mai 1984)

FORTGESCHRITTENE 6502-PROGRAMMIERUNG
von Rodnay Zaks – hilft Ihnen, schwierige Probleme mit dem 6502 zu lösen, stellt Ihnen Maschinenroutinen zum Arbeiten mit einem Hobbyboard vor. 288 Seiten, 140 Abbildungen, Ref.-Nr.: **3047** (April 1984)

COMMODORE 64 BASIC HANDBUCH
von D. Hergert – zeigt Ihnen alle Anwendungsmöglichkeiten Ihres C64 und beschreibt das vollständige BASIC-Vokabular anhand von praktischen Beispielen. Ca. 192 Seiten, ca. 100 Abbildungen, Ref.-Nr.: **3048** (April 1984)

PROGRAMMIERUNG DES 6809
von R. Zaks und W. Labiak – eine vollständige Einführung in die Assemblerprogrammierung mit dem 6809, für alle, die mit DRAGON 32, Tandy Colorcomputer, Commodore MMF 9000 oder einem anderen 6809-System arbeiten. Ca. 376 Seiten, 150 Abbildungen, Ref.-Nr.: **3049** (Mai 1984)

COMMODORE 64 PROGRAMMSAMMLUNG
von S. R. Trost – mehr als 70 getestete Anwenderprogramme, die direkt eingegeben werden können. Erläuterungen gewährleisten eine optimale Nutzung. 192 Seiten, 160 Abbildungen, Ref.-Nr.: **3051** (1983)

CP/M-HANDBUCH
von Rodnay Zaks – das Standardwerk über CP/M, das meistgebrauchte Betriebssystem für Mikrocomputer. Für Anfänger eine verständliche Einführung, für Fortgeschrittene ein umfassendes Nachschlagewerk über die CP/M-Versionen 2.2, 3.0 und CCP/M-86 sowie MP/M. 2., überarbeitete Ausgabe. Ca. 350 Seiten, ca. 100 Abbildungen, Ref.-Nr.: **3053** (1984)

UNIX-HANDBUCH
von R. Detering – eine systematische Einführung in UNIX, das kommende Betriebssystem für 16-bit-Rechner. Lernen Sie, Ihren Prozessor optimal einzusetzen! Ca. 280 Seiten, ca. 30 Abbildungen, Ref.-Nr.: **3054** (Mai 1984)

ERFOLGREICH PROGRAMMIEREN MIT C
von J. A. Illik – ein unentbehrliches Handbuch für jeden, der mit der universellen Sprache C erfolgreich programmieren will. Aussagekräftige Beispiele, auf verschiedenen Mini- und Mikrocomputern getestet. Ca. 250 Seiten, Ref.-Nr.: **3055** (Juni 1984)

ARBEITEN MIT DEM IBM PC
von J. Lasselle und C. Ramsay – zeigt Ihnen Schritt für Schritt, wie Sie den IBM PC ohne Vorkenntnisse einsetzen, die speziellen Eigenschaften dieses Computers für Druck, Grafik und Kommunikation nutzen können. Ca. 160 Seiten, ca. 30 Abbildungen, Ref.-Nr.: **3056** (Mai 1984)

PRAKTISCHE WORDSTAR-ANWENDUNGEN
von J. A. Arca – das Buch für Einsteiger, um nach kurzer Zeit praktische Textverarbeitungs-Probleme zu lösen, eine programmierte Unterweisung zur Leistungsoptimierung mit WORDSTAR. Ca. 320 Seiten, ca. 50 Abbildungen, Ref.-Nr.: **3057** (Juni 1984)

MEIN ERSTES COMMODORE 64-PROGRAMM
von R. Zaks – sollte Ihr erstes Buch zum Commodore 64 sein. Viel Spaß am Lernen durch farbige Illustrationen und leichtverständliche Diagramme, Programmieren mit sofortigen Resultaten. 208 Seiten, illustriert, Ref.-Nr.·**3062** (Mai 1984)

SYBEX MIKROCOMPUTER LEXIKON
– die schnelle Informationsbörse! Über 1500 Definitionen, Kurzformeln, Begriffsschema der Mikroprozessor-Technik, englisch/deutsches und französisch/deutsches Wörterbuch, Bezugsquellen. 192 Seiten, Format 12,5 x 18 cm, Ref.Nr.: **3005**

FORDERN SIE EIN GESAMTVERZEICHNIS UNSERER VERLAGSPRODUKTION AN:

SYBEX

SYBEX-VERLAG GmbH
Vogelsanger Weg 111
4000 Düsseldorf 30
Tel.: (02 11) 62 64 41
Telex: 8 588 163

SYBEX
6–8, Impasse du Curé
75018 Paris
Tel.: 1/203-95-95
Telex: 211.801 f

SYBEX INC.
2344 Sixth Street
Berkeley, CA 94710, USA
Tel.: (415) 848-8233
Telex: 336311

SYBEX

Programmierung des 8086/8088

Mit diesem Buch erlernen Sie die Programmierung, Kontrolle und Anwendung eines der mächtigsten und vielseitigsten 16-Bit-Mikroprozessoren, die heute erhältlich sind. Von der internen Architektur bis hin zu fortgeschrittenen Adressierungstechniken wird Ihnen das notwendige Wissen vermittelt, um Ihren 8086/8088-Mikroprozessor optimal nutzen zu können.

Sie lernen

- Grundkonzepte der Mikroprozessorprogrammierung
- die interne Struktur des 8086/8088
- die Speicherorganisation
- die komplette Struktur der Unterbrechungssignale
- Eingabe- und Ausgabetechniken
- die Nutzung des 8088 zur Beherrschung Ihres IBM PC u.v.a.m.

Anhand vieler Beispielprogramme werden die bereitgestellten Instruktionen leicht verständlich.

Spezifische Unterschiede sowie Ähnlichkeiten zwischen dem 8086 und dem 8088 werden im Detail erklärt.

Über den Autor

James W. Coffron ist Spezialist im Entwurf und in der Auswertung von Mikroprozessor-Computersystemen. Er erhielt s[redacted] of Science in Electrical Engineering von der Santa [redacted] sity Kalifornien. Über seine Lehrtätigkeit an Hoch[redacted] Seminare in der Industrie hinaus ist er Autor meh[redacted] über Mikroprozessoren.

ISBN 3-88745-[redacted]